var Routes = map[string]socket.CallBackFunc{

"block": GetBlock,

}

var logger log.Logger

func SetLogger(l log.Logger) {

logger = l

}

func GetBlock(req map[string]interface{}) (result interface{}, err error) {

height := req["height"].(int64)

if height < 0 {

err = errors.New(fmt.Sprintf("invalid height=%d", height))

return

}

if height == 0 {

appState := statedbhelper.GetWorldAppState(0, 0)

height = appState.BlockHeight

}

url := query.TmCoreURL

if url == "" {

err = errors.New("can not get tendermint url")

return nil, err

}

res := new(core\_types.ResultBlock)

rpc := rpcclient.NewJSONRPCClientEx(url, "", true)

\_, err = rpc.Call("block", map[string]interface{}{"height": height}, res)

if err != nil {

return nil, err

}

b := std.Block{

ChainID: res.BlockMeta.Header.ChainID,

BlockHash: types2.Hash(res.BlockMeta.BlockID.Hash),

Height: res.BlockMeta.Header.Height,

Time: res.BlockMeta.Header.Time.Unix(),

NumTxs: int32(res.BlockMeta.Header.NumTxs),

DataHash: types2.Hash(res.BlockMeta.Header.DataHash),

ProposerAddress: res.BlockMeta.Header.ProposerAddress,

RewardAddress: res.BlockMeta.Header.RewardAddress,

RandomNumber: types2.HexBytes(res.BlockMeta.Header.RandomOfBlock),

LastBlockHash: types2.Hash(res.BlockMeta.BlockID.Hash),

LastCommitHash: types2.Hash(res.BlockMeta.Header.LastCommitHash),

LastAppHash: types2.Hash(res.BlockMeta.Header.LastAppHash),

LastFee: int64(res.BlockMeta.Header.LastFee),

Version: \*res.BlockMeta.Header.Version,

}

resultByte, err := jsoniter.Marshal(b)

if err != nil {

return nil, err

}

result = string(resultByte)

return

}

func ConversionEventReceipt(logger log.Logger, transId, txId int64, res \*exec.LogEvent, EventReceipt \*LogEventParams, contractAddress types.Address, abiStr string) (name string, err error) {

newAbi := abi.ABI{}

if abiStr != "" {

newAbi, err = GetAbiObject(logger, abiStr)

if err != nil {

return

}

} else {

result := statedb.Get(transId, txId, "/gvm/contract/"+contractAddress)

if len(result) == 0 {

return "", err

}

contract := new(std.BvmContract)

err := json.Unmarshal(result, contract)

if err != nil {

panic("state db helper get gvm contract err: " + err.Error())

}

newAbi, err = GetAbiObject(logger, contract.BvmAbi)

if err != nil {

return "", err

}

}

for i := 0; i < len(newAbi.Events); i++ {

event, err := newAbi.EventByID(res.Topics[0].Bytes())

if err != nil {

return "", err

}

name = "::" + event.RawName

v := make(map[string]interface{})

err = newAbi.UnpackLogIntoMap(v, event.RawName, \*res)

EventReceipt.Data = make(map[string]interface{})

for \_, iv := range event.Inputs {

result := DetermineType(iv.Type.String(), v[iv.Name])

if iv.Indexed == true {

EventReceipt.Data["indexed/"+iv.Name] = result

} else {

EventReceipt.Data[iv.Name] = result

}

}

}

return

}

func DetermineType(paramType string, param interface{}) interface{} {

switch paramType {

case "bytes32":

bin := param.([32]byte)

return strings.Replace(string(bin[:]), "\u0000", "", -1)

case "bytes":

bin := param.([]byte)

return strings.Replace(string(bin[:]), "\u0000", "", -1)

case "byte[][]":

return GetBetweenStrByte(param.(string))

case "address":

addr := param.(abi.Address)

gvmAddr := crypto2.GVMAddress{}

copy(gvmAddr[:], addr[:])

address := crypto2.ToAddr(gvmAddr)

return address

case "address[]":

AddrSlice := make([]crypto.Address, 0)

slices := param.([]abi.Address)

for \_, v := range slices {

gvmAddr := crypto2.GVMAddress{}

copy(gvmAddr[:], v[:])

address := crypto2.ToAddr(gvmAddr)

AddrSlice = append(AddrSlice, address)

}

return AddrSlice

case "address[2]":

AddrSlice := make([]crypto.Address, 0)

slices := param.([2]abi.Address)

for \_, v := range slices {

gvmAddr := crypto2.GVMAddress{}

copy(gvmAddr[:], v[:])

address := crypto2.ToAddr(gvmAddr)

AddrSlice = append(AddrSlice, address)

}

return AddrSlice

case "address[][]":

return GetBetweenStrAddr(param.([][]abi.Address))

default:

return param

}

}

func GetBetweenStrByte(str string) (doubleSlice [][]byte) {

doubleSliceStr := make([][]string, 0)

reg := regexp.MustCompile(`\[(.\*?)\]`)

if reg != nil {

doubleSliceStr = reg.FindAllStringSubmatch(str, -1)

}

for i := 0; i < len(doubleSliceStr); i++ {

newStr := strings.Split(doubleSliceStr[i][1], ",")

for \_, v := range newStr {

doubleSlice = append(doubleSlice, []byte(v))

}

}

return

}

func GetBetweenStrAddr(addresses [][]abi.Address) (doubleSlice [][]crypto.Address) {

doubleSlice = make([][]crypto.Address, 0)

for i := 0; i < len(addresses); i++ {

newSlice := make([]crypto.Address, 0)

for \_, v := range addresses[i] {

gvmAddr := crypto2.GVMAddress{}

copy(gvmAddr[:], v[:])

address := crypto2.ToAddr(gvmAddr)

newSlice = append(newSlice, address)

}

doubleSlice = append(doubleSlice, newSlice)

}

return

}

// Emit emit receipt object

func Emit(logger log.Logger, receipt interface{}, contractAddress, abiStr string, idx int, transId, txId int64) \*common.KVPair {

if receipt == nil {

return nil

}

name, receiptObject := receiptName(receipt)

logger.Debug("receipt", "name", name, "res000", receiptObject)

EventReceipt := new(LogEventParams)

if name == "gvm::log\_event" {

EventName, err := ConversionEventReceipt(logger, transId, txId, receiptObject, EventReceipt, contractAddress, abiStr)

if err != nil {

logger.Debug("gvm", "ConversionEventReceiptErr", err)

}

name += EventName

}

Bz := make([]byte, 0)

if strings.HasPrefix(name, "gvm::log\_event") {

bz, err := jsoniter.Marshal(EventReceipt)

if err != nil {

sdkimpl.Logger.Fatalf("[sdk]Cannot marshal receipt data=%v", receipt)

sdkimpl.Logger.Flush()

panic(err)

}

Bz = append(Bz, bz...)

} else {

bz, err := jsoniter.Marshal(receipt)

if err != nil {

sdkimpl.Logger.Fatalf("[sdk]Cannot marshal receipt data=%v", receipt)

sdkimpl.Logger.Flush()

panic(err)

}

Bz = append(Bz, bz...)

}

rcpt := std.Receipt{

Name: name,

ContractAddr: contractAddress,

Bytes: Bz,

Hash: nil,

}

rcpt.Hash = sha3.Sum256([]byte(rcpt.Name), []byte(rcpt.ContractAddr), Bz)

resBytes, \_ := jsoniter.Marshal(rcpt) // nolint unhandled

//将收据添加到message

tag := common.KVPair{

Key: []byte(fmt.Sprintf("/0/%d/%s", idx, rcpt.Name)),

Value: resBytes,

}

return &tag

}

// receiptName - 收据名，转帐收据命名定死成标准转帐收据，不调用这个方法，其他收据就只有gvm事件了，加个gvm::，好认

func receiptName(receipt interface{}) (name string, event \*exec.LogEvent) {

switch r := receipt.(type) {

case \*exec.LogEvent:

op, \_ := receipt.(\*exec.LogEvent)

return "gvm::log\_event", op

case \*exec.CallEvent:

return "gvm::" + strings.ToLower(r.CallType.String()), nil

case \*exec.TransferData:

return "std::transfer", nil

case std.Fee:

return "std::fee", nil

default:

return "gvm::unknownType", nil

}

}

func Tags2Receipt(logger log.Logger, tags \*[]interface{}, transId, txId, fee int64, tokenAddr, contractAddr, sender, abiStr string) \*[]common.KVPair {

length := len(\*tags)

result := make([]common.KVPair, length+1, length+1)

for idx, t := range \*tags {

tag := Emit(logger, t, contractAddr, abiStr, idx+1, transId, txId)

result[idx+1] = \*tag

}

tag0 := Emit(logger,

std.Fee{

Token: tokenAddr,

From: sender,

Value: fee,

}, contractAddr, abiStr, 0, transId, txId)

result[0] = \*tag0

return &result

}

func GetGasPrice(transID, txID int64) int64 {

genToken := statedbhelper.GetGenesisToken().Address

genTokenCurrent := statedbhelper.GetTokenByAddress(transID, txID, genToken)

gasPriceRatio := statedbhelper.GetGasPriceRatio(transID, txID)

if gasPriceRatio == "" {

gasPriceRatio = "1.000"

}

gasPriceRatio = strings.Replace(gasPriceRatio, ".", "", -1)

uGasPriceRatio, err := strconv.ParseUint(gasPriceRatio, 10, 64)

if err != nil {

panic(err)

}

return genTokenCurrent.GasPrice \* int64(uGasPriceRatio) / 1000

}

//GetInstance get or create burrow instance

func GetInstance(log log.Logger) \*Burrow {

bu := &Burrow{}

bu.logger = log

bu.Tags = make([]interface{}, 0)

return bu

}

func (bu \*Burrow) InvokeTx(blockHeader types2.Header, blockHash []byte, transId, txId int64, sender types.Address, tx types.Transaction, pubKey types.PubKey) (result \*types.Response) {

gasPrice := receipt.GetGasPrice(transId, txId)

result = new(types.Response)

if IsCreate(tx.Messages) {

bu.logger.Debug("gvm: creating...")

result = bu.Create(blockHeader, blockHash, transId, txId, sender, tx, pubKey, gasPrice)

if e := checkBalanceForFee(transId, txId, result); e != nil {

result.Code = types.ErrFeeNotEnough

result.Log = e.Error()

result.Data = ""

}

return

} else if IsCall(tx.Messages) {

bu.logger.Debug("gvm: call...")

st := NewState(transId, txId, bu.logger)

contractAddr := tx.Messages[0].Contract

bu.logger.Debug("gvm:", "contractAddr", contractAddr)

account, err := st.GetAccount(crypto2.ToGVM(contractAddr))

if err != nil {

result = new(types.Response)

result.Code = types.ErrCodeGVMInvoke

result.Log = err.Error()

return

}

if account == nil {

result = new(types.Response)

result.Code = types.ErrCodeGVMInvoke

result.Log = "no such account"

return

}

bu.logger.Debug("gvm:", "contractToken", account.GVMToken, "code", hex.EncodeToString(account.GVMCode))

st.SetToken(account.GVMToken)

state := gvm.NewState(st, blockHashGetter)

result = bu.Call(state, blockHeader, blockHash, gasPrice, sender, account.GVMCode, tx, bn.N(0))

result.Tags = \*receipt.Tags2Receipt(bu.logger, &bu.Tags, transId, txId, result.GasUsed\*gasPrice, statedbhelper.GetGenesisToken().Address, contractAddr, sender, "")

if e := checkBalanceForFee(transId, txId, result); e != nil {

result.Code = types.ErrFeeNotEnough

result.Log = e.Error()

result.Data = ""

}

return

} else if IsCascadeCall(tx.Messages) {

bu.logger.Debug("gvm: cascadeCall...")

result = bu.CascadeCall(blockHeader, blockHash, transId, txId, sender, tx, gasPrice)

if e := checkBalanceForFee(transId, txId, result); e != nil {

result.Code = types.ErrFeeNotEnough

result.Log = e.Error()

result.Data = ""

}

return

} else {

result.Code = types.ErrCodeGVMInvoke

return

}

}

// Create of GVM contract create

func (bu \*Burrow) Create(blockHeader types2.Header, blockHash []byte, transId, txId int64, sender types.Address, tx types.Transaction, pubKey types.PubKey, gasPrice int64) (result \*types.Response) {

bu.logger.Debug("gvm:", "transId", transId, "txId", txId, "gas", tx.GasLimit)

bu.logger.Debug("gvm:", "tx", tx.String())

gas := uint64(tx.GasLimit) \* GasBvmRatio

result = new(types.Response)

nonce := make([]byte, 8)

sysbin.BigEndian.PutUint64(nonce, tx.Nonce)

st := NewState(transId, txId, bu.logger)

token := tx.Messages[0].Contract

tokenInfo := statedb.Get(transId, txId, std.KeyOfToken(token))

if len(tokenInfo) == 0 {

result.Code = types.ErrinvalidParam

result.Log = "token addr is not exits"

return

}

st.SetToken(token)

bu.logger.Debug("gvm:", "create contract token=", token)

state := gvm.NewState(st, blockHashGetter)

var code []byte

err := rlp.DecodeBytes(tx.Messages[0].Items[0], &code)

if err != nil {

result.Code = types.ErrinvalidParam

result.Log = err.Error()

return

} else if len(code) == 0 {

result.Code = types.ErrinvalidParam

result.Log = "gvm contract code could not be empty"

return

}

contractAddr := CalcContractAddr(sender, tx.Nonce, blockHeader.ChainID)

GVMAddr := crypto2.ToGVM(contractAddr)

state.CreateAccount(GVMAddr)

bu.logger.Debug("gvm:", "contractAddr", contractAddr, "gvmAddr", GVMAddr)

senderGVMAddr := crypto2.ToGVM(sender)

ourGVM := gvm.NewVM(newParams(blockHeader, blockHash, gasPrice, tx.GasLimit), senderGVMAddr, nonce, bu.logger)

out, err := ourGVM.Call(state, gvm.NewBcEventSink(bu.logger, &bu.Tags), senderGVMAddr, GVMAddr, code, nil, bn.N(0), &gas)

if err != nil {

result.Code = types.ErrCodeGVMInvoke

result.Log = err.Error()

return

}

var abiCode []byte

err = rlp.DecodeBytes(tx.Messages[0].Items[1], &abiCode)

if err != nil {

result.Code = types.ErrinvalidParam

result.Log = err.Error()

return

} else if len(abiCode) == 0 {

result.Code = types.ErrinvalidParam

result.Log = "contract's ABI info could not be empty"

return

}

abiStr := string(abiCode)

err = st.SetContractInfo(transId, txId, blockHeader.ChainVersion, token, crypto2.ToAddr(GVMAddr), crypto2.ToAddr(senderGVMAddr), abiStr)

if err != nil {

result.Code = types.ErrCodeGVMInvoke

result.Log = err.Error()

return

}

state.InitCode(GVMAddr, out)

if err2 := state.Sync(); err2 != nil {

result.Code = types.ErrCodeGVMInvoke

result.Log = err2.String()

return

}

result.Code = types.CodeOK

result.Data = contractAddr

result.GasUsed = (tx.GasLimit\*GasBvmRatio - int64(gas)) / GasBvmRatio

bu.logger.Debug("gvm:", "Create-GasUsed: ", result.GasUsed)

gasForCreate := int64(GasPerByte \* len(out))

bu.logger.Debug("gvm:", "bu.tags: ", bu.Tags)

result.Tags = \*receipt.Tags2Receipt(bu.logger, &bu.Tags, transId, txId,

(result.GasUsed+gasForCreate)\*gasPrice,

statedbhelper.GetGenesisToken().Address, contractAddr, sender, abiStr)

return

}

// Call of GVM contract contract call

func (bu \*Burrow) Call(state \*gvm.State, blockHeader types2.Header, blockHash []byte, gasPrice int64, sender types.Address, code []byte, tx types.Transaction, value bn.Number) (result \*types.Response) {

bu.logger.Debug("gvm:", "tx", tx.String())

gas := uint64(tx.GasLimit) \* GasBvmRatio

result = new(types.Response)

nonce := make([]byte, 8)

sysbin.BigEndian.PutUint64(nonce, 0)

var input []byte

err := rlp.DecodeBytes(tx.Messages[0].Items[0], &input)

if err != nil {

result.Code = types.ErrRlpDecode

result.Log = err.Error()

return

}

bu.logger.Debug("gvm:", "input", hex.EncodeToString(input))

senderGVMAddr := crypto2.ToGVM(sender)

ourGVM := gvm.NewVM(newParams(blockHeader, blockHash, gasPrice, tx.GasLimit), senderGVMAddr, nonce, bu.logger)

out, err := ourGVM.Call(state, gvm.NewBcEventSink(bu.logger, &bu.Tags), senderGVMAddr, crypto2.ToGVM(tx.Messages[0].Contract), code, input, value, &gas)

if err != nil {

result.Code = types.ErrCodeGVMInvoke

result.Log = err.Error()

return

}

if err2 := state.Sync(); err2 != nil {

result.Code = types.ErrCodeGVMInvoke

result.Log = err2.String()

return

}

isView, err := JudgeFuncType(tx.Messages[0].Contract, input[:4])

if isView {

result.Code = types.CodeGVMQueryOK

result.Data = hex.EncodeToString(out)

result.GasUsed = 0

} else {

result.Code = types.CodeOK

result.Data = hex.EncodeToString(out)

result.GasUsed = (tx.GasLimit\*GasBvmRatio - int64(gas)) / GasBvmRatio

bu.logger.Debug("gvm:", "Call-GasUsed: ", result.GasUsed)

}

return

}

// CascadeCall of GVM contract Cascade Call

func (bu \*Burrow) CascadeCall(blockHeader types2.Header, blockHash []byte, transId, txId int64, sender types.Address, tx types.Transaction, gasPrice int64) (result \*types.Response) {

contractAddr := tx.Messages[1].Contract

bu.logger.Debug("gvm:", "contractAddr", contractAddr)

st := NewState(transId, txId, bu.logger)

account, err := st.GetAccount(crypto2.ToGVM(contractAddr))

if err != nil {

result = new(types.Response)

result.Code = types.ErrCodeGVMInvoke

result.Log = err.Error()

return

}

if account == nil {

result = new(types.Response)

result.Code = types.ErrCodeGVMInvoke

result.Log = "no such account"

return

}

bu.logger.Debug("gvm:", "contractToken", account.GVMToken)

st.SetToken(account.GVMToken)

state := gvm.NewState(st, blockHashGetter)

var value []byte

err = rlp.DecodeBytes(tx.Messages[0].Items[0], &value)

if err != nil {

result = new(types.Response)

result.Code = types.ErrRlpDecode

result.Log = err.Error()

return

}

money := bn.NString(string(value)).MulI(1e9)

bu.logger.Debug("gvm:", "sender", sender, "paid", money)

// Contract call

tx.Messages = append(tx.Messages[1:])

result = bu.Call(state, blockHeader, blockHash, gasPrice, sender, account.GVMCode, tx, money)

result.Tags = \*receipt.Tags2Receipt(bu.logger, &bu.Tags, transId, txId, result.GasUsed\*gasPrice, statedbhelper.GetGenesisToken().Address, contractAddr, sender, "")

return

}

func newParams(blockHeader types2.Header, blockHash []byte, gasPrice, gasLimit int64) gvm.Params {

return gvm.Params{

BlockHeader: blockHeader,

GasLimit: uint64(gasLimit),

BlockHash: blockHash,

GasPrice: gasPrice,

CallStackMaxDepth: 0,

DataStackInitialCapacity: 0,

DataStackMaxDepth: 0,

}

}

func IsCreate(messages []types.Message) bool {

if len(messages) != 1 || messages[0].MethodID != 0 {

return false

}

return true

}

func IsCall(messages []types.Message) bool {

if len(messages) != 1 || messages[0].MethodID != 0xFFFFFFFF {

return false

}

return true

}

func IsCascadeCall(messages []types.Message) bool {

if len(messages) != 2 || messages[1].MethodID != 0xFFFFFFFF {

return false

}

return true

}

func blockHashGetter(height uint64) []byte {

req := make(map[string]interface{})

req["height"] = int64(height)

resp, err := burrowrpc.GetBlock(req)

if err != nil {

return nil

}

block := new(std.Block)

err = jsoniter.Unmarshal([]byte(resp.(string)), block)

if err != nil {

return nil

}

hash := block.BlockHash

return binary.LeftPadWord256(hash).Bytes()

}

func CalcContractAddr(senderAddr types.Address, nonce uint64, chainID string) types.Address {

nonceBin, err := rlp.EncodeToBytes(nonce)

if err != nil {

return ""

}

hasherSHA3256 := sha3.New256()

hasherSHA3256.Write([]byte(senderAddr))

hasherSHA3256.Write(nonceBin)

sha := hasherSHA3256.Sum(nil)

hasherRIPEMD160 := ripemd160.New()

hasherRIPEMD160.Write(sha) // does not error

rpd := hasherRIPEMD160.Sum(nil)

hasher := ripemd160.New()

hasher.Write(rpd)

md := hasher.Sum(nil)

addr := make([]byte, 0, len(rpd)+len(md[:4]))

addr = append(addr, rpd...)

addr = append(addr, md[:4]...)

return chainID + base58.Encode(addr)

}

func checkBalanceForFee(transId, txID int64, response \*types.Response) error {

fee := getFee(response.Tags)

senderBal := statedbhelper.BalanceOf(transId, txID, fee.From, fee.Token)

if senderBal.IsLessThanI(response.Fee) {

return errors.New("Insufficient balance to pay fee")

}

return nil

}

func getFee(tags []common.KVPair) std.Fee {

for \_, t := range tags {

if strings.Contains(string(t.Key), "std::fee") {

receipt := std.Receipt{}

err := jsoniter.Unmarshal(t.Value, &receipt)

if err != nil {

panic(err)

}

f := std.Fee{}

err = jsoniter.Unmarshal(receipt.Bytes, &f)

if err != nil {

panic(err)

}

return f

}

}

//panic("no fee receipt")

return std.Fee{}

}

// Determines whether the function call type is view

func JudgeFuncType(address types.Address, methodID []byte) (IsView bool, err error) {

contract := GetContractInfo(address)

Abi, err := receipt.GetAbiObject(nil, contract.BvmAbi)

if err != nil {

return

}

method, err := Abi.MethodById(methodID)

if err != nil {

return

}

IsView = method.Const

return

}

func (s \*State) UpdateAccount(updated \*acm.Account) error {

val, \_ := jsoniter.Marshal([]string{statedbhelper.KeyOfAccountNonce(crypto.ToAddr(updated.Address))})

statedb.Set(s.transID, s.txID, statedbhelper.KeyOfAccount(crypto.ToAddr(updated.Address)), val)

statedbhelper.SetGVMBalance(s.transID, s.txID, crypto.ToAddr(updated.Address), s.tokenAddr, updated.Balance)

if updated.GVMCode != nil {

statedb.Set(s.transID, s.txID, codeKey(updated.Address), updated.GVMCode)

if s.tokenAddr != "" {

statedb.Set(s.transID, s.txID, tokenKey(updated.Address), []byte(s.tokenAddr))

} else {

statedb.Set(s.transID, s.txID, tokenKey(updated.Address), []byte(updated.GVMToken))

}

}

err := updated.AddAccountTokenKey(s.transID, s.txID, std.KeyOfAccountToken(crypto.ToAddr(updated.Address), s.GetToken()))

if err != nil {

return err

}

return nil

}

func (s \*State) RemoveAccount(removed crypto.GVMAddress) error {

addr := crypto.ToAddr(removed)

statedb.Set(s.transID, s.txID, statedbhelper.KeyOfAccount(addr), nil)

statedb.Set(s.transID, s.txID, statedbhelper.KeyOfAccountNonce(addr), nil)

statedb.Set(s.transID, s.txID, codeKey(removed), nil)

statedb.Set(s.transID, s.txID, tokenKey(removed), nil)

return nil

}

func (s \*State) SetStorage(address crypto.GVMAddress, key binary.Word256, value []byte) error {

statedb.Set(s.transID, s.txID, storageKey(address, key), value)

return nil

}

func (s \*State) GetAccount(address crypto.GVMAddress) (\*acm.Account, error) {

addr := crypto.ToAddr(address)

s.logger.Debug("enter GetAccount:", "address", address, "stateToken", s.tokenAddr,

"transID", s.transID, "txID", s.txID)

if statedb.Get(s.transID, s.txID, statedbhelper.KeyOfAccount(addr)) == nil {

return nil, nil

}

act := acm.Account{Address: address}

balance := statedbhelper.GVMBalanceOf(s.transID, s.txID, addr, s.tokenAddr)

gvmToken := statedb.Get(s.transID, s.txID, tokenKey(address))

gvmCode := statedb.Get(s.transID, s.txID, codeKey(address))

act.Balance = balance

act.GVMToken = string(gvmToken)

act.GVMCode = gvmCode

s.logger.Debug("GetAccount return:", "account", act.String())

return &act, nil

}

func (s \*State) GetStorage(address crypto.GVMAddress, key binary.Word256) (value []byte, err error) {

byt := statedb.Get(s.transID, s.txID, storageKey(address, key))

return byt, nil

}

func (s \*State) SetContractInfo(transID, txID, ChainVersion int64, token, gvmAddr, senderAddr goCrypto.Address, abiStr string) (err error) {

if abiStr == "" {

s.logger.Debug("gvm", "abiFile is empty, please check")

return

}

abiStr = strings.Replace(abiStr, "\n", "", -1)

abiStr = strings.Replace(abiStr, "\t", "", -1)

abiStr = strings.Replace(abiStr, `\`, "", -1)

newAbi, err := abi.JSON(strings.NewReader(abiStr))

if err != nil {

return

}

Methods := make([]std.BvmMethod, 0)

Events := make([]string, 0)

for \_, v := range newAbi.Methods {

var Method std.BvmMethod

Method.MethodID = hex.EncodeToString(v.ID())

Method.ProtoType = v.ShortString()

Methods = append(Methods, Method)

}

for \_, v := range newAbi.Events {

Events = append(Events, v.ShortString())

}

GVMContract := new(std.BvmContract)

GVMContract.Methods = Methods

GVMContract.Events = Events

GVMContract.Address = gvmAddr

GVMContract.Token = token

GVMContract.Deployer = senderAddr

GVMContract.BvmAbi = abiStr

GVMContract.ChainVersion = ChainVersion

err = setGVMContract(transID, txID, GVMContract)

if err != nil {

s.logger.Debug("gvm", "set gvmContract Info to db failed")

return

}

return

}

func GetContractInfo(address goCrypto.Address) (contract \*std.BvmContract) {

res := statedb.Get(0, 0, contractInfoKey(address))

if len(res) == 0 {

return

}

contract = new(std.BvmContract)

if err := json.Unmarshal(res, &contract); err != nil {

return

}

return

}

func setGVMContract(transID, txID int64, contract \*std.BvmContract) error {

key := contractInfoKey(contract.Address)

value, err := jsoniter.Marshal(contract)

if err != nil {

panic(err)

}

statedb.Set(transID, txID, key, value)

return nil

}

type GVMType interface {

GetSignature() string

getGoType() interface{}

pack(v interface{}) ([]byte, error)

unpack(data []byte, offset int, v interface{}) (int, error)

Dynamic() bool

}

var \_ GVMType = (\*GVMBool)(nil)

type GVMBool struct {

}

func (e GVMBool) GetSignature() string {

return "bool"

}

func (e GVMBool) getGoType() interface{} {

return new(bool)

}

func (e GVMBool) pack(v interface{}) ([]byte, error) {

var b bool

arg := reflect.ValueOf(v)

if arg.Kind() == reflect.String {

val := arg.String()

if strings.EqualFold(val, "true") || val == "1" {

b = true

} else if strings.EqualFold(val, "false") || val == "0" {

b = false

} else {

return nil, fmt.Errorf("%s is not a valid value for GVM Bool type", val)

}

} else if arg.Kind() == reflect.Bool {

b = arg.Bool()

} else {

return nil, fmt.Errorf("%s cannot be converted to GVM Bool type", arg.Kind().String())

}

res := make([]byte, ElementSize)

if b {

res[ElementSize-1] = 1

}

return res, nil

}

func (e GVMBool) unpack(data []byte, offset int, v interface{}) (int, error) {

if len(data)-offset < 32 {

return 0, fmt.Errorf("not enough data")

}

data = data[offset:]

switch v := v.(type) {

case \*string:

if data[ElementSize-1] == 1 {

\*v = "true"

} else if data[ElementSize-1] == 0 {

\*v = "false"

} else {

return 0, fmt.Errorf("unexpected value for GVM bool")

}

case \*int8:

\*v = int8(data[ElementSize-1])

case \*int16:

\*v = int16(data[ElementSize-1])

case \*int32:

\*v = int32(data[ElementSize-1])

case \*int64:

\*v = int64(data[ElementSize-1])

case \*int:

\*v = int(data[ElementSize-1])

case \*uint8:

\*v = uint8(data[ElementSize-1])

case \*uint16:

\*v = uint16(data[ElementSize-1])

case \*uint32:

\*v = uint32(data[ElementSize-1])

case \*uint64:

\*v = uint64(data[ElementSize-1])

case \*uint:

\*v = uint(data[ElementSize-1])

case \*bool:

\*v = data[ElementSize-1] == 1

default:

return 0, fmt.Errorf("cannot set type %s for GVM bool", reflect.ValueOf(v).Kind().String())

}

return 32, nil

}

func (e GVMBool) Dynamic() bool {

return false

}

var \_ GVMType = (\*GVMUint)(nil)

type GVMUint struct {

M uint64

}

func (e GVMUint) GetSignature() string {

return fmt.Sprintf("uint%d", e.M)

}

func (e GVMUint) getGoType() interface{} {

switch e.M {

case 8:

return new(uint8)

case 16:

return new(uint16)

case 32:

return new(uint32)

case 64:

return new(uint64)

default:

return new(big.Int)

}

}

func (e GVMUint) pack(v interface{}) ([]byte, error) {

n := new(big.Int)

arg := reflect.ValueOf(v)

switch arg.Kind() {

case reflect.String:

\_, ok := n.SetString(arg.String(), 0)

if !ok {

return nil, fmt.Errorf("Failed to parse `%s", arg.String())

}

if n.Sign() < 0 {

return nil, fmt.Errorf("negative value not allowed for uint%d", e.M)

}

case reflect.Uint8:

fallthrough

case reflect.Uint16:

fallthrough

case reflect.Uint32:

fallthrough

case reflect.Uint64:

fallthrough

case reflect.Uint:

n.SetUint64(arg.Uint())

case reflect.Int8:

fallthrough

case reflect.Int16:

fallthrough

case reflect.Int32:

fallthrough

case reflect.Int64:

fallthrough

case reflect.Int:

x := arg.Int()

if x < 0 {

return nil, fmt.Errorf("negative value not allowed for uint%d", e.M)

}

n.SetInt64(x)

default:

t := reflect.TypeOf(new(uint64))

if reflect.TypeOf(v).ConvertibleTo(t) {

n.SetUint64(reflect.ValueOf(v).Convert(t).Uint())

} else {

return nil, fmt.Errorf("cannot convert type %s to uint%d", arg.Kind().String(), e.M)

}

}

b := n.Bytes()

if uint64(len(b)) > e.M {

return nil, fmt.Errorf("value to large for int%d", e.M)

}

return pad(b, ElementSize, true), nil

}

func (e GVMUint) unpack(data []byte, offset int, v interface{}) (int, error) {

if len(data)-offset < ElementSize {

return 0, fmt.Errorf("not enough data")

}

data = data[offset:]

empty := 0

for empty = 0; empty < ElementSize; empty++ {

if data[empty] != 0 {

break

}

}

length := ElementSize - empty

switch v := v.(type) {

case \*string:

b := new(big.Int)

b.SetBytes(data[empty:ElementSize])

\*v = b.String()

case \*big.Int:

b := new(big.Int)

\*v = \*b.SetBytes(data[0:ElementSize])

case \*uint64:

maxLen := int(unsafe.Sizeof(\*v))

if length > maxLen {

return 0, fmt.Errorf("value to large for uint64")

}

\*v = binary.BigEndian.Uint64(data[ElementSize-maxLen : ElementSize])

case \*uint32:

maxLen := int(unsafe.Sizeof(\*v))

if length > maxLen {

return 0, fmt.Errorf("value to large for uint64")

}

\*v = binary.BigEndian.Uint32(data[ElementSize-maxLen : ElementSize])

case \*uint16:

maxLen := int(unsafe.Sizeof(\*v))

if length > maxLen {

return 0, fmt.Errorf("value to large for uint16")

}

\*v = binary.BigEndian.Uint16(data[ElementSize-maxLen : ElementSize])

case \*uint8:

maxLen := 1

if length > maxLen {

return 0, fmt.Errorf("value to large for uint8")

}

\*v = uint8(data[31])

case \*int64:

maxLen := int(unsafe.Sizeof(\*v))

if length > maxLen || (data[ElementSize-maxLen]&0x80) != 0 {

return 0, fmt.Errorf("value to large for int64")

}

\*v = int64(binary.BigEndian.Uint64(data[ElementSize-maxLen : ElementSize]))

case \*int32:

maxLen := int(unsafe.Sizeof(\*v))

if length > maxLen || (data[ElementSize-maxLen]&0x80) != 0 {

return 0, fmt.Errorf("value to large for int64")

}

\*v = int32(binary.BigEndian.Uint32(data[ElementSize-maxLen : ElementSize]))

case \*int16:

maxLen := int(unsafe.Sizeof(\*v))

if length > maxLen || (data[ElementSize-maxLen]&0x80) != 0 {

return 0, fmt.Errorf("value to large for int16")

}

\*v = int16(binary.BigEndian.Uint16(data[ElementSize-maxLen : ElementSize]))

case \*int8:

maxLen := 1

if length > maxLen || (data[ElementSize-maxLen]&0x80) != 0 {

return 0, fmt.Errorf("value to large for int8")

}

\*v = int8(data[ElementSize-1])

default:

return 0, fmt.Errorf("unable to convert %s to %s", e.GetSignature(), reflect.ValueOf(v).Kind().String())

}

return 32, nil

}

func (e GVMUint) Dynamic() bool {

return false

}

var \_ GVMType = (\*GVMInt)(nil)

type GVMInt struct {

M uint64

}

func (e GVMInt) getGoType() interface{} {

switch e.M {

case 8:

return new(int8)

case 16:

return new(int16)

case 32:

return new(int32)

case 64:

return new(int64)

default:

return new(big.Int)

}

}

func (e GVMInt) GetSignature() string {

return fmt.Sprintf("int%d", e.M)

}

func (e GVMInt) pack(v interface{}) ([]byte, error) {

n := new(big.Int)

arg := reflect.ValueOf(v)

switch arg.Kind() {

case reflect.String:

\_, ok := n.SetString(arg.String(), 0)

if !ok {

return nil, fmt.Errorf("Failed to parse `%s", arg.String())

}

case reflect.Uint8:

fallthrough

case reflect.Uint16:

fallthrough

case reflect.Uint32:

fallthrough

case reflect.Uint64:

fallthrough

case reflect.Uint:

n.SetUint64(arg.Uint())

case reflect.Int8:

fallthrough

case reflect.Int16:

fallthrough

case reflect.Int32:

fallthrough

case reflect.Int64:

fallthrough

case reflect.Int:

n.SetInt64(arg.Int())

default:

t := reflect.TypeOf(new(int64))

if reflect.TypeOf(v).ConvertibleTo(t) {

n.SetInt64(reflect.ValueOf(v).Convert(t).Int())

} else {

return nil, fmt.Errorf("cannot convert type %s to int%d", arg.Kind().String(), e.M)

}

}

b := n.Bytes()

if uint64(len(b)) > e.M {

return nil, fmt.Errorf("value to large for int%d", e.M)

}

res := pad(b, ElementSize, true)

if (res[0] & 0x80) != 0 {

return nil, fmt.Errorf("value to large for int%d", e.M)

}

if n.Sign() < 0 {

// One's complement; i.e. 0xffff is -1, not 0.

n.Add(n, big.NewInt(1))

b := n.Bytes()

res = pad(b, ElementSize, true)

for i := 0; i < len(res); i++ {

res[i] = ^res[i]

}

}

return res, nil

}

func (e GVMInt) unpack(data []byte, offset int, v interface{}) (int, error) {

if len(data)-offset < ElementSize {

return 0, fmt.Errorf("not enough data")

}

data = data[offset:]

sign := (data[0] & 0x80) != 0

empty := 0

for empty = 0; empty < ElementSize; empty++ {

if (sign && data[empty] != 255) || (!sign && data[empty] != 0) {

break

}

}

length := ElementSize - empty

inv := make([]byte, ElementSize)

for i := 0; i < ElementSize; i++ {

if sign {

inv[i] = ^data[i]

} else {

inv[i] = data[i]

}

}

toType := reflect.ValueOf(v).Kind().String()

switch v := v.(type) {

case \*string:

b := new(big.Int)

b.SetBytes(inv[empty:ElementSize])

if sign {

\*v = b.Sub(big.NewInt(-1), b).String()

} else {

\*v = b.String()

}

case \*big.Int:

b := new(big.Int)

b.SetBytes(inv[0:ElementSize])

if sign {

\*v = \*b.Sub(big.NewInt(-1), b)

} else {

\*v = \*b

}

case \*uint64:

if sign {

return 0, fmt.Errorf("cannot convert negative GVM int to %s", toType)

}

maxLen := int(unsafe.Sizeof(\*v))

if length > maxLen {

return 0, fmt.Errorf("value to large for uint64")

}

\*v = binary.BigEndian.Uint64(data[ElementSize-maxLen : ElementSize])

case \*uint32:

if sign {

return 0, fmt.Errorf("cannot convert negative GVM int to %s", toType)

}

maxLen := int(unsafe.Sizeof(\*v))

if length > maxLen {

return 0, fmt.Errorf("value to large for int32")

}

\*v = binary.BigEndian.Uint32(data[ElementSize-maxLen : ElementSize])

case \*uint16:

if sign {

return 0, fmt.Errorf("cannot convert negative GVM int to %s", toType)

}

maxLen := int(unsafe.Sizeof(\*v))

if length > maxLen {

return 0, fmt.Errorf("value to large for uint16")

}

\*v = binary.BigEndian.Uint16(data[ElementSize-maxLen : ElementSize])

case \*int64:

maxLen := int(unsafe.Sizeof(\*v))

if length > maxLen || (inv[ElementSize-maxLen]&0x80) != 0 {

return 0, fmt.Errorf("value to large for int64")

}

\*v = int64(binary.BigEndian.Uint64(data[ElementSize-maxLen : ElementSize]))

case \*int32:

maxLen := int(unsafe.Sizeof(\*v))

if length > maxLen || (inv[ElementSize-maxLen]&0x80) != 0 {

return 0, fmt.Errorf("value to large for uint64")

}

\*v = int32(binary.BigEndian.Uint32(data[ElementSize-maxLen : ElementSize]))

case \*int16:

maxLen := int(unsafe.Sizeof(\*v))

if length > maxLen || (inv[ElementSize-maxLen]&0x80) != 0 {

return 0, fmt.Errorf("value to large for uint16")

}

\*v = int16(binary.BigEndian.Uint16(data[ElementSize-maxLen : ElementSize]))

default:

return 0, fmt.Errorf("unable to convert %s to %s", e.GetSignature(), toType)

}

return ElementSize, nil

}

func (e GVMInt) Dynamic() bool {

return false

}

var \_ GVMType = (\*GVMAddress)(nil)

type GVMAddress struct {

}

func (e GVMAddress) String() string {

return "GVMAddress"

}

func (e GVMAddress) getGoType() interface{} {

return new(crypto.GVMAddress)

}

func (e GVMAddress) GetSignature() string {

return "address"

}

func (e GVMAddress) pack(v interface{}) ([]byte, error) {

var err error

a, ok := v.(crypto.GVMAddress)

if !ok {

s, ok := v.(string)

if ok {

a, err = crypto.AddressFromHexString(s)

if err != nil {

return nil, err

}

}

} else {

b, ok := v.([]byte)

if !ok {

return nil, fmt.Errorf("cannot map to %s to GVM address", reflect.ValueOf(v).Kind().String())

}

a, err = crypto.AddressFromBytes(b)

if err != nil {

return nil, err

}

}

return pad(a[:], ElementSize, true), nil

}

func (e GVMAddress) Dynamic() bool {

return false

}

func (e GVMBytes) pack(v interface{}) ([]byte, error) {

b, ok := v.([]byte)

if !ok {

s, ok := v.(string)

if ok {

b = []byte(s)

} else {

return nil, fmt.Errorf("cannot map to %s to GVM bytes", reflect.ValueOf(v).Kind().String())

}

}

if e.M > 0 {

if uint64(len(b)) > e.M {

return nil, fmt.Errorf("[%d]byte to long for %s", len(b), e.GetSignature())

}

return pad(b, ElementSize, false), nil

} else {

length := GVMUint{M: 256}

p, err := length.pack(len(b))

if err != nil {

return nil, err

}

for i := 0; i < len(b); i += ElementSize {

a := b[i:]

if len(a) == 0 {

break

}

p = append(p, pad(a, ElementSize, false)...)

}

return p, nil

}

}

func (e GVMBytes) unpack(data []byte, offset int, v interface{}) (int, error) {

if e.M == 0 {

s := GVMString{}

return s.unpack(data, offset, v)

}

v2 := reflect.ValueOf(v).Elem()

switch v2.Type().Kind() {

case reflect.String:

start := 0

end := int(e.M)

for start < ElementSize-1 && data[offset+start] == 0 && start < end {

start++

}

for end > start && data[offset+end-1] == 0 {

end--

}

v2.SetString(string(data[offset+start : offset+end]))

case reflect.Array:

fallthrough

case reflect.Slice:

v2.SetBytes(data[offset : offset+int(e.M)])

default:

return 0, fmt.Errorf("cannot map GVM %s to %s", e.GetSignature(), reflect.ValueOf(v).Kind().String())

}

return ElementSize, nil

}

func (e GVMBytes) Dynamic() bool {

return e.M == 0

}

func (e GVMBytes) GetSignature() string {

if e.M > 0 {

return fmt.Sprintf("bytes%d", e.M)

} else {

return "bytes"

}

}

var \_ GVMType = (\*GVMString)(nil)

type GVMString struct {

}

func (e GVMString) GetSignature() string {

return "string"

}

func (e GVMString) getGoType() interface{} {

return new(string)

}

func (e GVMString) pack(v interface{}) ([]byte, error) {

b := GVMBytes{M: 0}

return b.pack(v)

}

func (e GVMString) unpack(data []byte, offset int, v interface{}) (int, error) {

lenType := GVMInt{M: 64}

var len int64

l, err := lenType.unpack(data, offset, &len)

if err != nil {

return 0, err

}

offset += l

switch v := v.(type) {

case \*string:

\*v = string(data[offset : offset+int(len)])

case \*[]byte:

\*v = data[offset : offset+int(len)]

default:

return 0, fmt.Errorf("cannot map GVM string to %s", reflect.ValueOf(v).Kind().String())

}

return ElementSize, nil

}

func (e GVMString) Dynamic() bool {

return true

}

var \_ GVMType = (\*GVMFixed)(nil)

type GVMFixed struct {

N, M uint64

signed bool

}

func (e GVMFixed) getGoType() interface{} {

// This is not right, obviously

return new(big.Float)

}

func (e GVMFixed) GetSignature() string {

if e.signed {

return fmt.Sprintf("fixed%dx%d", e.M, e.N)

} else {

return fmt.Sprintf("ufixed%dx%d", e.M, e.N)

}

}

func (e GVMFixed) pack(v interface{}) ([]byte, error) {

// The ABI spec does not describe how this should be packed; go-ethereum abi does not implement this

// need to dig in solidity to find out how this is packed

return nil, fmt.Errorf("packing of %s not implemented, patches welcome", e.GetSignature())

}

func (e GVMFixed) unpack(data []byte, offset int, v interface{}) (int, error) {

// The ABI spec does not describe how this should be packed; go-ethereum abi does not implement this

// need to dig in solidity to find out how this is packed

return 0, fmt.Errorf("unpacking of %s not implemented, patches welcome", e.GetSignature())

}

func (e GVMFixed) Dynamic() bool {

return false

}

type Argument struct {

Name string

GVM GVMType

IsArray bool

Indexed bool

Hashed bool

ArrayLength uint64

}

const FunctionIDSize = 4

type FunctionID [FunctionIDSize]byte

const EventIDSize = 32

type EventID [EventIDSize]byte

type FunctionSpec struct {

FunctionID FunctionID

Constant bool

Inputs []Argument

Outputs []Argument

}

type EventSpec struct {

EventID EventID

Inputs []Argument

Name string

Anonymous bool

}

type AbiSpec struct {

Constructor FunctionSpec

Fallback FunctionSpec

Functions map[string]FunctionSpec

Events map[string]EventSpec

EventsById map[EventID]EventSpec

}

type ArgumentJSON struct {

Name string

Type string

Components []ArgumentJSON

Indexed bool

}

type AbiSpecJSON struct {

Name string

Type string

Inputs []ArgumentJSON

Outputs []ArgumentJSON

Constant bool

Payable bool

StateMutability string

Anonymous bool

}

func readArgSpec(argsJ []ArgumentJSON) ([]Argument, error) {

args := make([]Argument, len(argsJ))

var err error

for i, a := range argsJ {

args[i].Name = a.Name

args[i].Indexed = a.Indexed

baseType := a.Type

isArray := regexp.MustCompile(`(.\*)\[([0-9]+)\]`)

m := isArray.FindStringSubmatch(a.Type)

if m != nil {

args[i].IsArray = true

args[i].ArrayLength, err = strconv.ParseUint(m[2], 10, 32)

if err != nil {

return nil, err

}

baseType = m[1]

} else if strings.HasSuffix(a.Type, "[]") {

args[i].IsArray = true

baseType = strings.TrimSuffix(a.Type, "[]")

}

isM := regexp.MustCompile("(bytes|uint|int)([0-9]+)")

m = isM.FindStringSubmatch(baseType)

if m != nil {

M, err := strconv.ParseUint(m[2], 10, 32)

if err != nil {

return nil, err

}

switch m[1] {

case "bytes":

if M < 1 || M > 32 {

return nil, fmt.Errorf("bytes%d is not valid type", M)

}

args[i].GVM = GVMBytes{M}

case "uint":

if M < 8 || M > 256 || (M%8) != 0 {

return nil, fmt.Errorf("uint%d is not valid type", M)

}

args[i].GVM = GVMUint{M}

case "int":

if M < 8 || M > 256 || (M%8) != 0 {

return nil, fmt.Errorf("uint%d is not valid type", M)

}

args[i].GVM = GVMInt{M}

}

continue

}

isMxN := regexp.MustCompile("(fixed|ufixed)([0-9]+)x([0-9]+)")

m = isMxN.FindStringSubmatch(baseType)

if m != nil {

M, err := strconv.ParseUint(m[2], 10, 32)

if err != nil {

return nil, err

}

N, err := strconv.ParseUint(m[3], 10, 32)

if err != nil {

return nil, err

}

if M < 8 || M > 256 || (M%8) != 0 {

return nil, fmt.Errorf("%s is not valid type", baseType)

}

if N == 0 || N > 80 {

return nil, fmt.Errorf("%s is not valid type", baseType)

}

if m[1] == "fixed" {

args[i].GVM = GVMFixed{N: N, M: M, signed: true}

} else if m[1] == "ufixed" {

args[i].GVM = GVMFixed{N: N, M: M, signed: false}

} else {

panic(m[1])

}

continue

}

switch baseType {

case "uint":

args[i].GVM = GVMUint{M: 256}

case "int":

args[i].GVM = GVMInt{M: 256}

case "address":

args[i].GVM = GVMAddress{}

case "bool":

args[i].GVM = GVMBool{}

case "fixed":

args[i].GVM = GVMFixed{M: 128, N: 8, signed: true}

case "ufixed":

args[i].GVM = GVMFixed{M: 128, N: 8, signed: false}

case "bytes":

args[i].GVM = GVMBytes{M: 0}

case "string":

args[i].GVM = GVMString{}

default:

// Assume it is a type of Contract

args[i].GVM = GVMAddress{}

}

}

return args, nil

}

func ReadAbiSpec(specBytes []byte) (\*AbiSpec, error) {

var specJ []AbiSpecJSON

err := json.Unmarshal(specBytes, &specJ)

if err != nil {

// The abi spec file might a bin file, with the Abi under the Abi field in json

var binFile struct {

Abi []AbiSpecJSON

}

err = json.Unmarshal(specBytes, &binFile)

if err != nil {

return nil, err

}

specJ = binFile.Abi

}

abiSpec := AbiSpec{

Events: make(map[string]EventSpec),

EventsById: make(map[EventID]EventSpec),

Functions: make(map[string]FunctionSpec),

}

for \_, s := range specJ {

switch s.Type {

case "constructor":

abiSpec.Constructor.Inputs, err = readArgSpec(s.Inputs)

if err != nil {

return nil, err

}

case "fallback":

abiSpec.Fallback.Inputs = make([]Argument, 0)

abiSpec.Fallback.Outputs = make([]Argument, 0)

case "event":

inputs, err := readArgSpec(s.Inputs)

if err != nil {

return nil, err

}

// Get signature before we deal with hashed types

sig := Signature(s.Name, inputs)

for i := range inputs {

if inputs[i].Indexed && inputs[i].GVM.Dynamic() {

// For Dynamic types, the hash is stored in stead

inputs[i].GVM = GVMBytes{M: 32}

inputs[i].Hashed = true

}

}

ev := EventSpec{Name: s.Name, EventID: GetEventID(sig), Inputs: inputs, Anonymous: s.Anonymous}

abiSpec.Events[ev.Name] = ev

abiSpec.EventsById[ev.EventID] = ev

case "function":

inputs, err := readArgSpec(s.Inputs)

if err != nil {

return nil, err

}

outputs, err := readArgSpec(s.Outputs)

if err != nil {

return nil, err

}

fs := FunctionSpec{Inputs: inputs, Outputs: outputs, Constant: s.Constant}

fs.SetFunctionID(s.Name)

abiSpec.Functions[s.Name] = fs

}

}

return &abiSpec, nil

}

// MergeAbiSpec takes multiple AbiSpecs and merges them into once structure. Note that

// the same function name or event name can occur in different abis, so there might be

// some information loss.

func MergeAbiSpec(abiSpec []\*AbiSpec) \*AbiSpec {

newSpec := AbiSpec{

Events: make(map[string]EventSpec),

EventsById: make(map[EventID]EventSpec),

Functions: make(map[string]FunctionSpec),

}

for \_, s := range abiSpec {

for n, f := range s.Functions {

newSpec.Functions[n] = f

}

// Different Abis can have the Event name, but with a different signature

// Loop over the signatures, as these are less likely to have collisions

for \_, e := range s.EventsById {

newSpec.Events[e.Name] = e

newSpec.EventsById[e.EventID] = e

}

}

return &newSpec

}

func GVMTypeFromReflect(v reflect.Type) Argument {

arg := Argument{Name: v.Name()}

if v == reflect.TypeOf("") {

arg.GVM = GVMAddress{}

} else if v == reflect.TypeOf(big.Int{}) {

arg.GVM = GVMInt{M: 256}

} else {

if v.Kind() == reflect.Array {

arg.IsArray = true

arg.ArrayLength = uint64(v.Len())

v = v.Elem()

} else if v.Kind() == reflect.Slice {

arg.IsArray = true

v = v.Elem()

}

switch v.Kind() {

case reflect.Bool:

arg.GVM = GVMBool{}

case reflect.String:

arg.GVM = GVMString{}

case reflect.Uint64:

arg.GVM = GVMUint{M: 64}

case reflect.Int64:

arg.GVM = GVMInt{M: 64}

default:

panic(fmt.Sprintf("no mapping for type %v", v.Kind()))

}

}

return arg

}

// SpecFromStructReflect generates a FunctionSpec where the arguments and return values are

// described a struct. Both args and rets should be set to the return value of reflect.TypeOf()

// with the respective struct as an argument.

func SpecFromStructReflect(fname string, args reflect.Type, rets reflect.Type) \*FunctionSpec {

s := FunctionSpec{

Inputs: make([]Argument, args.NumField()),

Outputs: make([]Argument, rets.NumField()),

}

for i := 0; i < args.NumField(); i++ {

f := args.Field(i)

a := GVMTypeFromReflect(f.Type)

a.Name = f.Name

s.Inputs[i] = a

}

for i := 0; i < rets.NumField(); i++ {

f := rets.Field(i)

a := GVMTypeFromReflect(f.Type)

a.Name = f.Name

s.Outputs[i] = a

}

s.SetFunctionID(fname)

return &s

}

func SpecFromFunctionReflect(fname string, v reflect.Value, skipIn, skipOut int) \*FunctionSpec {

t := v.Type()

if t.Kind() != reflect.Func {

panic(fmt.Sprintf("%s is not a function", t.Name()))

}

s := FunctionSpec{}

s.Inputs = make([]Argument, t.NumIn()-skipIn)

s.Outputs = make([]Argument, t.NumOut()-skipOut)

for i := range s.Inputs {

s.Inputs[i] = GVMTypeFromReflect(t.In(i + skipIn))

}

for i := range s.Outputs {

s.Outputs[i] = GVMTypeFromReflect(t.Out(i))

}

s.SetFunctionID(fname)

return &s

}

func Signature(name string, args []Argument) (sig string) {

sig = name + "("

for i, a := range args {

if i > 0 {

sig += ","

}

sig += a.GVM.GetSignature()

if a.IsArray {

if a.ArrayLength > 0 {

sig += fmt.Sprintf("[%d]", a.ArrayLength)

} else {

sig += "[]"

}

}

}

sig += ")"

return

}

func (functionSpec \*FunctionSpec) SetFunctionID(functionName string) {

sig := Signature(functionName, functionSpec.Inputs)

functionSpec.FunctionID = GetFunctionID(sig)

}

func (fs FunctionID) Bytes() []byte {

return fs[:]

}

func GetFunctionID(signature string) (id FunctionID) {

hash := sha3.NewKeccak256()

hash.Write([]byte(signature))

copy(id[:], hash.Sum(nil)[:4])

return

}

func (fs EventID) Bytes() []byte {

return fs[:]

}

func GetEventID(signature string) (id EventID) {

hash := sha3.NewKeccak256()

hash.Write([]byte(signature))

copy(id[:], hash.Sum(nil))

return

}

// UnpackRevert decodes the revert reason if a contract called revert. If no

// reason was given, message will be nil else it will point to the string

func UnpackRevert(data []byte) (message \*string, err error) {

if len(data) > 0 {

var msg string

err = RevertAbi.UnpackWithID(data, &msg)

message = &msg

}

return

}

/\*

\* Given a eventSpec, get all the fields (topic fields or not)

\*/

func UnpackEvent(eventSpec \*EventSpec, topics []burrowBinary.Word256, data []byte, args ...interface{}) error {

// First unpack the topic fields

topicIndex := 0

if !eventSpec.Anonymous {

topicIndex++

}

for i, a := range eventSpec.Inputs {

if a.Indexed {

\_, err := a.GVM.unpack(topics[topicIndex].Bytes(), 0, args[i])

if err != nil {

return err

}

topicIndex++

}

}

// Now unpack the other fields. unpack will step over any indexed fields

return unpack(eventSpec.Inputs, data, func(i int) interface{} {

return args[i]

})

}

func (abiSpec \*AbiSpec) Unpack(data []byte, fname string, args ...interface{}) error {

var funcSpec FunctionSpec

var argSpec []Argument

if fname != "" {

if \_, ok := abiSpec.Functions[fname]; ok {

funcSpec = abiSpec.Functions[fname]

} else {

funcSpec = abiSpec.Fallback

}

} else {

funcSpec = abiSpec.Constructor

}

argSpec = funcSpec.Outputs

if argSpec == nil {

return fmt.Errorf("Unknown function %s", fname)

}

return unpack(argSpec, data, func(i int) interface{} {

return args[i]

})

}

func (abiSpec \*AbiSpec) UnpackWithID(data []byte, args ...interface{}) error {

var argSpec []Argument

var id FunctionID

copy(id[:], data)

for \_, fspec := range abiSpec.Functions {

if id == fspec.FunctionID {

argSpec = fspec.Outputs

}

}

if argSpec == nil {

return fmt.Errorf("Unknown function %x", id)

}

return unpack(argSpec, data[4:], func(i int) interface{} {

return args[i]

})

}

// Pack ABI encodes a function call. The fname specifies which function should called, if

// it doesn't exist exist the fallback function will be called. If fname is the empty

// string, the constructor is called. The arguments must be specified in args. The count

// must match the function being called.

// Returns the ABI encoded function call, whether the function is constant according

// to the ABI (which means it does not modified contract state)

func (abiSpec \*AbiSpec) Pack(fname string, args ...interface{}) ([]byte, \*FunctionSpec, error) {

var funcSpec FunctionSpec

var argSpec []Argument

if fname != "" {

if \_, ok := abiSpec.Functions[fname]; ok {

funcSpec = abiSpec.Functions[fname]

} else {

funcSpec = abiSpec.Fallback

}

} else {

funcSpec = abiSpec.Constructor

}

argSpec = funcSpec.Inputs

if argSpec == nil {

if fname == "" {

return nil, nil, fmt.Errorf("Contract does not have a constructor")

}

return nil, nil, fmt.Errorf("Unknown function %s", fname)

}

packed := make([]byte, 0)

if fname != "" {

packed = funcSpec.FunctionID[:]

}

packedArgs, err := Pack(argSpec, args...)

if err != nil {

return nil, nil, err

}

return append(packed, packedArgs...), &funcSpec, nil

}

func PackIntoStruct(argSpec []Argument, st interface{}) ([]byte, error) {

v := reflect.ValueOf(st)

fields := v.NumField()

if fields != len(argSpec) {

return nil, fmt.Errorf("%d arguments expected, %d received", len(argSpec), fields)

}

return pack(argSpec, func(i int) interface{} {

return v.Field(i).Interface()

})

}

func Pack(argSpec []Argument, args ...interface{}) ([]byte, error) {

if len(args) != len(argSpec) {

return nil, fmt.Errorf("%d arguments expected, %d received", len(argSpec), len(args))

}

return pack(argSpec, func(i int) interface{} {

return args[i]

})

}

func pack(argSpec []Argument, getArg func(int) interface{}) ([]byte, error) {

packed := make([]byte, 0)

packedDynamic := []byte{}

fixedSize := 0

// Anything dynamic is stored after the "fixed" block. For the dynamic types, the fixed

// block contains byte offsets to the data. We need to know the length of the fixed

// block, so we can calcute the offsets

for \_, a := range argSpec {

if a.IsArray {

if a.ArrayLength > 0 {

fixedSize += ElementSize \* int(a.ArrayLength)

} else {

fixedSize += ElementSize

}

} else {

fixedSize += ElementSize

}

}

addArg := func(v interface{}, a Argument) error {

var b []byte

var err error

if a.GVM.Dynamic() {

offset := GVMUint{M: 256}

b, \_ = offset.pack(fixedSize)

d, err := a.GVM.pack(v)

if err != nil {

return err

}

fixedSize += len(d)

packedDynamic = append(packedDynamic, d...)

} else {

b, err = a.GVM.pack(v)

if err != nil {

return err

}

}

packed = append(packed, b...)

return nil

}

for i, as := range argSpec {

a := getArg(i)

if as.IsArray {

s, ok := a.(string)

if ok && s[0:1] == "[" && s[len(s)-1:] == "]" {

a = strings.Split(s[1:len(s)-1], ",")

}

val := reflect.ValueOf(a)

if val.Kind() != reflect.Slice && val.Kind() != reflect.Array {

return nil, fmt.Errorf("argument %d should be array or slice, not %s", i, val.Kind().String())

}

if as.ArrayLength > 0 {

if as.ArrayLength != uint64(val.Len()) {

return nil, fmt.Errorf("argumment %d should be array of %d, not %d", i, as.ArrayLength, val.Len())

}

for n := 0; n < val.Len(); n++ {

err := addArg(val.Index(n).Interface(), as)

if err != nil {

return nil, err

}

}

} else {

// dynamic array

offset := GVMUint{M: 256}

b, \_ := offset.pack(fixedSize)

packed = append(packed, b...)

fixedSize += len(b)

// store length

b, \_ = offset.pack(val.Len())

packedDynamic = append(packedDynamic, b...)

for n := 0; n < val.Len(); n++ {

d, err := as.GVM.pack(val.Index(n).Interface())

if err != nil {

return nil, err

}

packedDynamic = append(packedDynamic, d...)

}

}

} else {

err := addArg(a, as)

if err != nil {

return nil, err

}

}

}

return append(packed, packedDynamic...), nil

}

func GetPackingTypes(args []Argument) []interface{} {

res := make([]interface{}, len(args))

for i, a := range args {

if a.IsArray {

t := reflect.TypeOf(a.GVM.getGoType())

res[i] = reflect.MakeSlice(reflect.SliceOf(t), int(a.ArrayLength), 0).Interface()

} else {

res[i] = a.GVM.getGoType()

}

}

return res

}

func UnpackIntoStruct(argSpec []Argument, data []byte, st interface{}) error {

v := reflect.ValueOf(st).Elem()

return unpack(argSpec, data, func(i int) interface{} {

return v.Field(i).Addr().Interface()

})

}

func Unpack(argSpec []Argument, data []byte, args ...interface{}) error {

return unpack(argSpec, data, func(i int) interface{} {

return args[i]

})

}

func unpack(argSpec []Argument, data []byte, getArg func(int) interface{}) error {

offset := 0

offType := GVMInt{M: 64}

getPrimitive := func(e interface{}, a Argument) error {

if a.GVM.Dynamic() {

var o int64

l, err := offType.unpack(data, offset, &o)

if err != nil {

return err

}

offset += l

\_, err = a.GVM.unpack(data, int(o), e)

if err != nil {

return err

}

} else {

l, err := a.GVM.unpack(data, offset, e)

if err != nil {

return err

}

offset += l

}

return nil

}

for i, a := range argSpec {

if a.Indexed {

continue

}

arg := getArg(i)

if a.IsArray {

var array \*[]interface{}

array, ok := arg.(\*[]interface{})

if !ok {

if \_, ok := arg.(\*string); ok {

// We have been asked to return the value as a string; make intermediate

// array of strings; we will concatenate after

intermediate := make([]interface{}, a.ArrayLength)

for i := range intermediate {

intermediate[i] = new(string)

}

array = &intermediate

} else {

return fmt.Errorf("argument %d should be array, slice or string", i)

}

}

if a.ArrayLength > 0 {

if int(a.ArrayLength) != len(\*array) {

return fmt.Errorf("argument %d should be array or slice of %d elements", i, a.ArrayLength)

}

for n := 0; n < len(\*array); n++ {

err := getPrimitive((\*array)[n], a)

if err != nil {

return err

}

}

} else {

var o int64

var length int64

l, err := offType.unpack(data, offset, &o)

if err != nil {

return err

}

offset += l

s, err := offType.unpack(data, int(o), &length)

if err != nil {

return err

}

o += int64(s)

intermediate := make([]interface{}, length)

if \_, ok := arg.(\*string); ok {

// We have been asked to return the value as a string; make intermediate

// array of strings; we will concatenate after

for i := range intermediate {

intermediate[i] = new(string)

}

} else {

for i := range intermediate {

intermediate[i] = a.GVM.getGoType()

}

}

for i := 0; i < int(length); i++ {

l, err = a.GVM.unpack(data, int(o), intermediate[i])

if err != nil {

return err

}

o += int64(l)

}

array = &intermediate

}

// If we were supposed to return a string, convert it back

if ret, ok := arg.(\*string); ok {

s := "["

for i, e := range \*array {

if i > 0 {

s += ","

}

s += \*(e.(\*string))

}

s += "]"

\*ret = s

}

} else {

err := getPrimitive(arg, a)

if err != nil {

return err

}

}

}

return nil

}

// quick helper padding

func pad(input []byte, size int, left bool) []byte {

if len(input) >= size {

return input[:size]

}

padded := make([]byte, size)

if left {

copy(padded[size-len(input):], input)

} else {

copy(padded, input)

}

return padded

}

func (es \*BcEventSink) Call(call \*exec.CallEvent, exception \*errors.Exception) error {

es.Logger.Debug("gvmCallEvent:", call)

es.Logger.Debug("gvmException:", exception)

// CallEvent

\*es.Tags = append(\*es.Tags, call)

// Exception

if exception != nil {

\*es.Tags = append(\*es.Tags, exception)

}

return nil

}

func (es \*BcEventSink) CallTransfer(transfer \*exec.TransferData, exception \*errors.Exception) error {

es.Logger.Debug("gvmTransferEvent:", transfer)

es.Logger.Debug("gvmException:", exception)

// CallTransfer

\*es.Tags = append(\*es.Tags, transfer)

// Exception

if exception != nil {

\*es.Tags = append(\*es.Tags, exception)

}

return nil

}

func (es \*BcEventSink) Log(log \*exec.LogEvent) error {

es.Logger.Debug("gvmLogEvent:", log)

// LogEvent

\*es.Tags = append(\*es.Tags, log)

return nil

}

type logFreeEventSink struct {

EventSink

}

func NewLogFreeEventSink(eventSink EventSink) \*logFreeEventSink {

return &logFreeEventSink{

EventSink: eventSink,

}

}

func (esc \*logFreeEventSink) Log(log \*exec.LogEvent) error {

return errors.ErrorCodef(errors.ErrorCodeIllegalWrite,

"Log emitted from contract %v, but current call should be log-free", log.Address)

}

const (

TransferTypeCall = TransferType(0x00)

)

var nameFromTransferType = map[TransferType]string{

TransferTypeCall: "Transfer",

}

var TransferTypeFromName = make(map[string]TransferType)

func init() {

for t, n := range nameFromTransferType {

TransferTypeFromName[n] = t

}

}

func TransferTypeFromString(name string) TransferType {

return TransferTypeFromName[name]

}

func (tt TransferType) String() string {

name, ok := nameFromTransferType[tt]

if ok {

return name

}

return "UnknownTransferType"

}

func (tt TransferType) MarshalText() ([]byte, error) {

return []byte(tt.String()), nil

}

func (tt \*TransferType) UnmarshalText(data []byte) error {

\*tt = TransferTypeFromString(string(data))

return nil

}

type TransferData struct {

Token crypto2.Address `json:"token"` // Token types.Address

From crypto2.Address `json:"from"` // Account address of Sender

To crypto2.Address `json:"to"` // Account address of Receiver

Value bn.Number `json:"value"` // Transfer value

Note string `json:"note,omitempty"` // Transfer note

}

func (m \*TransferData) String() string {

return fmt.Sprintf("{Caller: %s, Callee: %s, Data: %s, Value: %s, Gas: %d}",

m.Token,

m.From,

m.To,

m.Value.String(),

m.Note)

}

func (m \*TransferData) GetValue() bn.Number {

if m != nil {

return m.Value

}

return bn.N(0)

}

func (m \*TransferData) GetNote() string {

if m != nil {

return m.Note

}

return ""

}

type TransferEvent struct {

TransferData \*TransferData `json:"TransferData,omitempty"`

}