```
pragma solidity ^0.4.11;

contract ERC20 {
  function totalSupply() constant returns (uint totalSupply);
  function balanceOf(address _owner) constant returns (uint balance);
  function transfer(address _to, uint _value) returns (bool success);
  function transferFrom(address _from, address _to, uint _value) returns (bool success);
  function approve(address _spender, uint _value) returns (bool success);
  function allowance(address _owner, address _spender) constant returns (uint remaining);
  event Transfer(address indexed _from, address indexed _to, uint _value);
  event Approval(address indexed _owner, address indexed _spender, uint _value);
}

// Limit order book with an ERC20 token as base, and ETH as quoted.
//
contract BookERC20EthV1 {

  enum BookType {
    ERC20EthV1
  }

  enum Direction {
    Invalid,
    Buy,
    Sell
  }

  enum Status {
    Unknown,
    Rejected,
    Open,
    Done,
    NeedsGas,
    Sending, // not used by contract - web only
    FailedSend // not used by contract - web only
  }

  enum ReasonCode {
    None,
    InvalidPrice,
    InvalidSize,
    InvalidTerms,
    InsufficientFunds,
    WouldTake,
    Unmatched,
    TooManyMatches,
    ClientCancel
  }

  enum Terms {
    GTCNoGasTopup,
    GTCWithGasTopup,
    ImmediateOrCancel,
    MakerOnly
  }

  struct Order {
    // these are immutable once placed:
```

```
  address client;
  uint16 price;           // packed representation of side + price
  uint sizeBase;
  Terms terms;

  // these are mutable until Done or Rejected:

  Status status;
  ReasonCode reasonCode;
  uint executedBase;      // gross amount executed (that is, before fee deduction)
  uint executedCntr;      // gross amount executed (that is, before fee deduction)
  uint fees;              // fees charged in base for a buy order, in counter for a sell order
}

struct OrderChain {
  uint128 firstOrderId;
  uint128 lastOrderId;
}

struct OrderChainNode {
  uint128 nextOrderId;
  uint128 prevOrderId;
}

enum ClientPaymentEventType {
  Deposit,
  Withdraw,
  TransferFrom,
  Transfer,
  Approve
}

enum BaseOrCntr {
  Base,
  Cntr
}

event ClientPaymentEvent(
  address indexed client,
  ClientPaymentEventType clientPaymentEventType,
  BaseOrCntr baseOrCntr,
  int clientBalanceDelta
);

enum ClientOrderEventType {
  Create,
  Continue,
  Cancel
}

event ClientOrderEvent(
  address indexed client,
  ClientOrderEventType clientOrderEventType,
  uint128 orderId
);

enum MarketOrderEventType {
```

```
  Add,
  Remove,
  CompleteFill,
  PartialFill
}

event MarketOrderEvent(
  uint128 indexed orderId,
  MarketOrderEventType marketOrderEventType,
  uint16 price,
  uint amountBase
);

// the base token (e.g. UBI)

ERC20 baseToken;
uint constant baseMinInitialSize = 100; // yes, far too small - testing only!
// if following partial match, the remaning gets smaller than this, remove from book and refund:
uint constant baseMinRemainingSize = 10; // yes, far too small - testing only!
uint constant baseMaxSize = 2 ** 127;

// the counter currency (ETH)

// no address because it is ETH
uint constant cntrMinInitialSize = 10000; // yes, far too small - testing only!
// only base has min remaining size
uint constant cntrMaxSize = 2 ** 127;

// funds that belong to clients

mapping (address => uint) balanceBaseForClient;
mapping (address => uint) balanceCntrForClient;

// fee charged on liquidity taken in parts-per-million

uint constant feePpm = 500;

// fees charged are given to:

address investorProxy;

// all orders ever created

mapping (uint128 => Order) orderForOrderId;

// Effectively a compact mapping from price to whether there are any open orders at that price.
// See "Price Calculation Constants" below as to why 85.

uint256[85] occupiedPriceBitmaps;

// These allow us to walk over the orders in the book at a given price level (and add more).

mapping (uint16 => OrderChain) orderChainForOccupiedPrice;
mapping (uint128 => OrderChainNode) orderChainNodeForOpenOrderId;

// These allow a client to (reasonably) efficiently find their own orders
// without relying on events (which even indexed are a bit expensive to search
// and cannot be accessed from smart contracts). See walkOrders.
```

```
mapping (address => uint128) mostRecentOrderIdForClient;
mapping (uint128 => uint128) clientPreviousOrderIdBeforeOrderId;

// Price Calculation Constants.
//
// We pack direction and price into a crafty decimal floating point representation
// for efficient indexing by price, the main thing we lose by doing so is precision -
// we only have 3 significant figures in our prices.
//
// An unpacked price consists of:
//
//   direction - invalid / buy / sell
//   mantissa  - ranges from 100 to 999 representing 0.100 to 0.999
//   exponent  - ranges from minimumPriceExponent to minimumPriceExponent + 11
//               (e.g. -5 to +6 for a typical pair where minPriceExponent = -5)
//
// The packed representation has 21601 different price values:
//
//      0  = invalid (can be used as marker value)
//      1  = buy at maximum price (0.999 * 10 ** 6)
//    ...  = other buy prices in descending order
//   5401  = buy at 1.00
//    ...  = other buy prices in descending order
//  10800  = buy at minimum price (0.100 * 10 ** -5)
//  10801  = sell at minimum price (0.100 * 10 ** -5)
//    ...  = other sell prices in descending order
//  16201  = sell at 1.00
//    ...  = other sell prices in descending order
//  21600  = sell at maximum price (0.999 * 10 ** 6)
//  21601+ = do not use
//
// If we want to map each packed price to a boolean value (which we do),
// we require 85 256-bit words. Or 42.5 for each side of the book.

int8 constant minPriceExponent = -5;

uint constant invalidPrice = 0;

// careful: max = largest unpacked value, not largest packed value
uint constant maxBuyPrice = 1;
uint constant minBuyPrice = 10800;
uint constant minSellPrice = 10801;
uint constant maxSellPrice = 21600;

// Constructor.
//
// Sets investorProxy to the creator. Creator needs to call init() to finish setup.
//
function BookERC20EthV1() {
  address creator = msg.sender;
  investorProxy = creator;
}

// "Public" Management - set address of base token.
//
// Can only be done once (normally immediately after creation) by the investor proxy.
//
```

```
// Used instead of a constructor to make deployment easier.
//
function init(ERC20 _baseToken) public {
  require(msg.sender == investorProxy);
  require(address(baseToken) == 0);
  require(address(_baseToken) != 0);
  // attempt to catch bad tokens:
  require(_baseToken.totalSupply() > 0);
  baseToken = _baseToken;
}

// "Public" Management - change investor proxy.
//
// The new investor proxy only gets fees charged after this point.
//
function changeInvestorProxy(address newInvestorProxy) public {
  address oldInvestorProxy = investorProxy;
  require(msg.sender == oldInvestorProxy);
  require(newInvestorProxy != oldInvestorProxy);
  investorProxy = newInvestorProxy;
}

// Public Info View - what is being traded here, what are the limits?
//
function getBookInfo() public constant returns (
    BookType _bookType,
    address _baseToken, uint _baseMinInitialSize,
    address _cntrToken, uint _cntrMinInitialSize,
    uint _feePpm, address _investorProxy
  ) {
  return (
    BookType.ERC20EthV1,
    address(baseToken),
    baseMinInitialSize,
    address(0),
    cntrMinInitialSize,
    feePpm,
    investorProxy
  );
}

// Public Funds View - get balances held by contract on behalf of the client.
//
// Excludes funds in open orders.
//
function getClientBalances(address client) public constant returns (uint balanceBase, uint balanceCntr) {
  return (balanceBaseForClient[client], balanceCntrForClient[client]);
}

// Public Funds Manipulation - deposit previously-approved base tokens.
//
function transferFromBase() public {
  address client = msg.sender;
  address book = address(this);
  uint amountBase = baseToken.allowance(client, book);
  require(amountBase > 0);
  require(baseToken.transferFrom(client, book, amountBase));
  assert(baseToken.allowance(client, book) == 0);
```

```
  balanceBaseForClient[client] += amountBase;
  ClientPaymentEvent(client, ClientPaymentEventType.TransferFrom, BaseOrCntr.Base, int(amountBase));
}

// Public Funds Manipulation - withdraw base tokens (as a transfer).
//
function transferBase(uint amountBase) public {
  address client = msg.sender;
  require(amountBase > 0);
  require(amountBase <= balanceBaseForClient[client]);
  balanceBaseForClient[client] -= amountBase;
  require(baseToken.transfer(client, amountBase));
  ClientPaymentEvent(client, ClientPaymentEventType.Transfer, BaseOrCntr.Base, -int(amountBase));
}

// Public Funds Manipulation - approve client to spend their base tokens.
//
// This probably only makes sense when the client is a smart-contract.
//
function approveBase(uint newAllowanceBase) public {
  address client = msg.sender;
  address book = address(this);
  uint oldAllowanceBase = baseToken.allowance(book, client);
  uint amountBase;
  if (newAllowanceBase > oldAllowanceBase) {
    amountBase = newAllowanceBase - oldAllowanceBase;
    require(amountBase <= balanceBaseForClient[client]);
    balanceBaseForClient[client] -= amountBase;
    require(baseToken.approve(client, newAllowanceBase));
    assert(baseToken.allowance(book, client) == newAllowanceBase);
    ClientPaymentEvent(client, ClientPaymentEventType.Approve, BaseOrCntr.Base, -int(amountBase));
  } else if (newAllowanceBase == oldAllowanceBase) {
    return;
  } else {
    amountBase = oldAllowanceBase - newAllowanceBase;
    require(baseToken.approve(client, newAllowanceBase));
    assert(baseToken.allowance(book, client) == newAllowanceBase);
    balanceBaseForClient[client] += amountBase;
    ClientPaymentEvent(client, ClientPaymentEventType.Approve, BaseOrCntr.Base, int(amountBase));
  }
}

// Public Funds Manipulation - deposit counter currency (ETH).
//
function depositCntr() public payable {
  address client = msg.sender;
  uint amountCntr = msg.value;
  require(amountCntr > 0);
  balanceCntrForClient[client] += amountCntr;
  ClientPaymentEvent(client, ClientPaymentEventType.Deposit, BaseOrCntr.Cntr, int(amountCntr));
}

// Public Funds Manipulation - withdraw counter currency (ETH).
//
function withdrawCntr(uint amountCntr) public {
  address client = msg.sender;
  require(amountCntr > 0);
  require(amountCntr <= balanceCntrForClient[client]);
```

```
  balanceCntrForClient[client] -= amountCntr;
  client.transfer(amountCntr);
  ClientPaymentEvent(client, ClientPaymentEventType.Withdraw, BaseOrCntr.Cntr, -int(amountCntr));
}

// Public Order View - get full details of an order.
//
// If the orderId does not exist, status will be Unknown.
//
function getOrder(uint128 orderId) public constant returns (
  address client, uint16 price, uint sizeBase, Terms terms,
  Status status, ReasonCode reasonCode, uint executedBase, uint executedCntr, uint fees) {
  Order order = orderForOrderId[orderId];
  return (order.client, order.price, order.sizeBase, order.terms,
          order.status, order.reasonCode, order.executedBase, order.executedCntr, order.fees);
}

// Public Order View - get mutable details of an order.
//
// If the orderId does not exist, status will be Unknown.
//
function getOrderState(uint128 orderId) public constant returns (
  Status status, ReasonCode reasonCode, uint executedBase, uint executedCntr, uint fees) {
  Order order = orderForOrderId[orderId];
  return (order.status, order.reasonCode, order.executedBase, order.executedCntr, order.fees);
}

// Public Order View - enumerate all recent orders + all open orders for one client.
//
// Not really designed for use from a smart contract transaction.
//
// Idea is:
//  - client ensures order ids are generated so that most-signficant part is time-based;
//  - client decides they want all orders after a certain point-in-time,
//    and chooses minClosedOrderIdCutoff accordingly;
//  - before that point-in-time they just get open and needs gas orders
//  - client calls walkClientOrders with maybeLastOrderIdReturned = 0 initially;
//  - then repeats with the orderId returned by walkClientOrders;
//  - (and stops if it returns a zero orderId);
//
// Note that client is only used when maybeLastOrderIdReturned = 0.
//
function walkClientOrders(
    address client, uint128 maybeLastOrderIdReturned, uint128 minClosedOrderIdCutoff
  ) public constant returns (
    uint128 orderId, uint16 price, uint sizeBase, Terms terms,
    Status status, ReasonCode reasonCode, uint executedBase, uint executedCntr, uint fees
  ) {
  if (maybeLastOrderIdReturned == 0) {
    orderId = mostRecentOrderIdForClient[client];
  } else {
    orderId = clientPreviousOrderIdBeforeOrderId[maybeLastOrderIdReturned];
  }
  while (true) {
    if (orderId == 0) return;
    Order order = orderForOrderId[orderId];
    if (orderId >= minClosedOrderIdCutoff) break;
    if (order.status == Status.Open || order.status == Status.NeedsGas) break;
```

```
    orderId = clientPreviousOrderIdBeforeOrderId[orderId];
  }
  return (orderId, order.price, order.sizeBase, order.terms,
          order.status, order.reasonCode, order.executedBase, order.executedCntr, order.fees);
}

// Internal Price Calculation - turn packed price into a friendlier unpacked price.
//
function unpackPrice(uint16 price) internal constant returns (
    Direction direction, uint16 mantissa, int8 exponent
  ) {
  uint sidedPriceIndex = uint(price);
  uint priceIndex;
  if (sidedPriceIndex < 1 || sidedPriceIndex > maxSellPrice) {
    direction = Direction.Invalid;
    mantissa = 0;
    exponent = 0;
    return;
  } else if (sidedPriceIndex <= minBuyPrice) {
    direction = Direction.Buy;
    priceIndex = minBuyPrice - sidedPriceIndex;
  } else {
    direction = Direction.Sell;
    priceIndex = sidedPriceIndex - minSellPrice;
  }
  uint zeroBasedMantissa = priceIndex % 900;
  uint zeroBasedExponent = priceIndex / 900;
  mantissa = uint16(zeroBasedMantissa + 100);
  exponent = int8(zeroBasedExponent) + minPriceExponent;
  return;
}

// Internal Price Calculation - is a packed price on the buy side?
//
// Throws an error if price is invalid.
//
function isBuyPrice(uint16 price) internal constant returns (bool isBuy) {
  // yes, this looks odd, but max here is highest _unpacked_ price
  return price >= maxBuyPrice && price <= minBuyPrice;
}

// Internal Price Calculation - turn a packed buy price into a packed sell price.
//
// Invalid price remains invalid.
//
function computeOppositePrice(uint16 price) internal constant returns (uint16 opposite) {
  if (price < maxBuyPrice || price > maxSellPrice) {
    return uint16(invalidPrice);
  } else if (price <= minBuyPrice) {
    return uint16(maxSellPrice - (price - maxBuyPrice));
  } else {
    return uint16(maxBuyPrice + (maxSellPrice - price));
  }
}

// Internal Price Calculation - compute amount in counter currency that would
// be obtained by selling baseAmount at the given unpacked price (if no fees).
//
```

```
// Notes:
//  - Does not validate price - caller must ensure valid.
//  - Could overflow producing very unexpected results if baseAmount very
//    large - caller must check this.
//  - This rounds the amount towards zero.
//  - May truncate to zero if baseAmount very small - potentially allowing
//    zero-cost buys or pointless sales - caller must check this.
//
function computeCntrAmountUsingUnpacked(
    uint baseAmount, uint16 mantissa, int8 exponent
  ) internal constant returns (uint cntrAmount) {
  if (exponent < 0) {
    return baseAmount * uint(mantissa) / 1000 / 10 ** uint(-exponent);
  } else {
    return baseAmount * uint(mantissa) / 1000 * 10 ** uint(exponent);
  }
}

// Internal Price Calculation - compute amount in counter currency that would
// be obtained by selling baseAmount at the given packed price (if no fees).
//
// Notes:
//  - Does not validate price - caller must ensure valid.
//  - Direction of the packed price is ignored.
//  - Could overflow producing very unexpected results if baseAmount very
//    large - caller must check this.
//  - This rounds the amount towards zero (regardless of Buy or Sell).
//  - May truncate to zero if baseAmount very small - potentially allowing
//    zero-cost buys or pointless sales - caller must check this.
//
function computeCntrAmountUsingPacked(
    uint baseAmount, uint16 price
  ) internal constant returns (uint) {
  var (, mantissa, exponent) = unpackPrice(price);
  return computeCntrAmountUsingUnpacked(baseAmount, mantissa, exponent);
}

// Public Order Placement - create order and try to match it and/or add it to the book.
//
function createOrder(
    uint128 orderId, uint16 price, uint sizeBase, Terms terms, uint maxMatches
  ) public {
  address client = msg.sender;
  if (client == 0 || orderId == 0 || orderForOrderId[orderId].client != 0) {
    throw;
  }
  ClientOrderEvent(client, ClientOrderEventType.Create, orderId);
  orderForOrderId[orderId] =
    Order(client, price, sizeBase, terms, Status.Unknown, ReasonCode.None, 0, 0, 0);
  uint128 previousMostRecentOrderIdForClient = mostRecentOrderIdForClient[client];
  mostRecentOrderIdForClient[client] = orderId;
  clientPreviousOrderIdBeforeOrderId[orderId] = previousMostRecentOrderIdForClient;
  Order order = orderForOrderId[orderId];
  var (direction, mantissa, exponent) = unpackPrice(price);
  if (direction == Direction.Invalid) {
    order.status = Status.Rejected;
    order.reasonCode = ReasonCode.InvalidPrice;
    return;
```

```
  }
  if (sizeBase < baseMinInitialSize || sizeBase > baseMaxSize) {
    order.status = Status.Rejected;
    order.reasonCode = ReasonCode.InvalidSize;
    return;
  }
  uint sizeCntr = computeCntrAmountUsingUnpacked(sizeBase, mantissa, exponent);
  if (sizeCntr < cntrMinInitialSize || sizeCntr > cntrMaxSize) {
    order.status = Status.Rejected;
    order.reasonCode = ReasonCode.InvalidSize;
    return;
  }
  if (terms == Terms.MakerOnly && maxMatches != 0) {
    order.status = Status.Rejected;
    order.reasonCode = ReasonCode.InvalidTerms;
    return;
  }
  if (!debitFunds(client, direction, sizeBase, sizeCntr)) {
    order.status = Status.Rejected;
    order.reasonCode = ReasonCode.InsufficientFunds;
    return;
  }
  processOrder(orderId, maxMatches);
}

// Public Order Placement - cancel order
//
function cancelOrder(uint128 orderId) public {
  address client = msg.sender;
  Order order = orderForOrderId[orderId];
  require(order.client == client);
  Status status = order.status;
  if (status != Status.Open && status != Status.NeedsGas) {
    return;
  }
  if (status == Status.Open) {
    removeOpenOrderFromBook(orderId);
    MarketOrderEvent(orderId, MarketOrderEventType.Remove, order.price,
      order.sizeBase - order.executedBase);
  }
  refundUnmatchedAndFinish(orderId, Status.Done, ReasonCode.ClientCancel);
}

// Public Order Placement - continue placing an order in 'NeedsGas' state
//
function continueOrder(uint128 orderId, uint maxMatches) public {
  address client = msg.sender;
  Order order = orderForOrderId[orderId];
  if (order.client != client) {
    throw;
  }
  if (order.status != Status.NeedsGas) {
    return;
  }
  order.status = Status.Unknown;
  processOrder(orderId, maxMatches);
}
```

```
// Internal Order Placement - remove a still-open order from the book.
//
// Caller's job to update/refund the order + raise event, this just
// updates the order chain and bitmask.
//
// Too expensive to do on each resting order match - we only do this for an
// order being cancelled. See matchWithOccupiedPrice for similar logic.
//
function removeOpenOrderFromBook(uint128 orderId) internal {
  Order order = orderForOrderId[orderId];
  uint16 price = order.price;
  OrderChain orderChain = orderChainForOccupiedPrice[price];
  OrderChainNode orderChainNode = orderChainNodeForOpenOrderId[orderId];
  uint128 nextOrderId = orderChainNode.nextOrderId;
  uint128 prevOrderId = orderChainNode.prevOrderId;
  if (nextOrderId != 0) {
    OrderChainNode nextOrderChainNode = orderChainNodeForOpenOrderId[nextOrderId];
    nextOrderChainNode.prevOrderId = prevOrderId;
  } else {
    orderChain.lastOrderId = prevOrderId;
  }
  if (prevOrderId != 0) {
    OrderChainNode prevOrderChainNode = orderChainNodeForOpenOrderId[prevOrderId];
    prevOrderChainNode.nextOrderId = nextOrderId;
  } else {
    orderChain.firstOrderId = nextOrderId;
  }
  if (nextOrderId == 0 && prevOrderId == 0) {
    uint bmi = price / 256;  // index into array of bitmaps
    uint bti = price % 256;  // bit position within bitmap
    // we know was previously occupied so XOR clears
    occupiedPriceBitmaps[bmi] ^= 2 ** bti;
  }
}

// Internal Order Placement - process a created and sanity checked order.
//
function processOrder(uint128 orderId, uint maxMatches) internal {
  Order order = orderForOrderId[orderId];

  uint ourOriginalExecutedBase = order.executedBase;
  uint ourOriginalExecutedCntr = order.executedCntr;
  var (ourDirection,) = unpackPrice(order.price);
  uint theirPriceStart = (ourDirection == Direction.Buy) ? minSellPrice : maxBuyPrice;
  uint theirPriceEnd = computeOppositePrice(order.price);

  MatchStopReason matchStopReason =
    matchAgainstBook(orderId, theirPriceStart, theirPriceEnd, maxMatches);

  uint liquidityTaken;
  uint fees;
  if (isBuyPrice(order.price)) {
    liquidityTaken = (order.executedBase - ourOriginalExecutedBase);
    if (liquidityTaken > 0) {
      fees = liquidityTaken * feePpm / 1000000;
      balanceBaseForClient[order.client] += (liquidityTaken - fees);
      order.fees += fees;
      balanceBaseForClient[investorProxy] += fees;
```

```
    }
  } else {
    liquidityTaken = (order.executedCntr - ourOriginalExecutedCntr);
    if (liquidityTaken > 0) {
      fees = liquidityTaken * feePpm / 1000000;
      balanceCntrForClient[order.client] += (liquidityTaken - fees);
      order.fees += fees;
      balanceCntrForClient[investorProxy] += fees;
    }
  }

  if (order.terms == Terms.ImmediateOrCancel) {
    if (matchStopReason == MatchStopReason.Satisfied) {
      refundUnmatchedAndFinish(orderId, Status.Done, ReasonCode.None);
      return;
    } else if (matchStopReason == MatchStopReason.MaxMatches) {
      refundUnmatchedAndFinish(orderId, Status.Done, ReasonCode.TooManyMatches);
      return;
    } else if (matchStopReason == MatchStopReason.BookExhausted) {
      refundUnmatchedAndFinish(orderId, Status.Done, ReasonCode.Unmatched);
      return;
    }
  } else if (order.terms == Terms.MakerOnly) {
    if (matchStopReason == MatchStopReason.MaxMatches) {
      refundUnmatchedAndFinish(orderId, Status.Rejected, ReasonCode.WouldTake);
      return;
    } else if (matchStopReason == MatchStopReason.BookExhausted) {
      enterOrder(orderId);
      return;
    }
  } else if (order.terms == Terms.GTCNoGasTopup) {
    if (matchStopReason == MatchStopReason.Satisfied) {
      refundUnmatchedAndFinish(orderId, Status.Done, ReasonCode.None);
      return;
    } else if (matchStopReason == MatchStopReason.MaxMatches) {
      refundUnmatchedAndFinish(orderId, Status.Done, ReasonCode.TooManyMatches);
      return;
    } else if (matchStopReason == MatchStopReason.BookExhausted) {
      enterOrder(orderId);
      return;
    }
  } else if (order.terms == Terms.GTCWithGasTopup) {
    if (matchStopReason == MatchStopReason.Satisfied) {
      refundUnmatchedAndFinish(orderId, Status.Done, ReasonCode.None);
      return;
    } else if (matchStopReason == MatchStopReason.MaxMatches) {
      order.status = Status.NeedsGas;
      return;
    } else if (matchStopReason == MatchStopReason.BookExhausted) {
      enterOrder(orderId);
      return;
    }
  }
  throw;
}

// Used internally to indicate why we stopped matching an order against the book.
```

```
enum MatchStopReason {
  None,
  MaxMatches,
  Satisfied,
  PriceExhausted,
  BookExhausted
}

// Internal Order Placement - Match the given order against the book.
//
// Resting orders matched will be updated, removed from book and funds credited to their owners.
//
// Only updates the executedBase and executedCntr of the given order - caller is responsible
// for crediting matched funds, charging fees, marking order as done / entering it into the book.
//
function matchAgainstBook(
    uint128 orderId, uint theirPriceStart, uint theirPriceEnd, uint maxMatches
  ) internal returns (
    MatchStopReason matchStopReason
  ) {
  Order order = orderForOrderId[orderId];

  uint bmi = theirPriceStart / 256;  // index into array of bitmaps
  uint bti = theirPriceStart % 256;  // bit position within bitmap
  uint bmiEnd = theirPriceEnd / 256; // last bitmap to search
  uint btiEnd = theirPriceEnd % 256; // stop at this bit in the last bitmap

  uint cbm = occupiedPriceBitmaps[bmi]; // original copy of current bitmap
  uint dbm = cbm; // dirty version of current bitmap where we may have cleared bits
  uint wbm = cbm >> bti; // working copy of current bitmap which we keep shifting

  // these loops are pretty ugly, and somewhat unpredicatable in terms of gas,
  // ... but no-one else has come up with a better matching engine yet!

  bool removedLastAtPrice;
  matchStopReason = MatchStopReason.None;

  while (bmi < bmiEnd) {
    if (wbm == 0 || bti == 256) {
      if (dbm != cbm) {
        occupiedPriceBitmaps[bmi] = dbm;
      }
      bti = 0;
      bmi++;
      cbm = occupiedPriceBitmaps[bmi];
      wbm = cbm;
      dbm = cbm;
    } else {
      if ((wbm & 1) != 0) {
        // careful - copy-and-pasted in loop below ...
        (removedLastAtPrice, maxMatches, matchStopReason) =
          matchWithOccupiedPrice(order, uint16(bmi * 256 + bti), maxMatches);
        if (removedLastAtPrice) {
          dbm ^= 2 ** bti;
        }
        if (matchStopReason == MatchStopReason.PriceExhausted) {
          matchStopReason = MatchStopReason.None;
        } else if (matchStopReason != MatchStopReason.None) {
```

```
          break;
        }
      }
      bti += 1;
      wbm /= 2;
    }
  }
  if (matchStopReason == MatchStopReason.None) {
    while (bti <= btiEnd && wbm != 0) {
      if ((wbm & 1) != 0) {
        // careful - copy-and-pasted in loop above ...
        (removedLastAtPrice, maxMatches, matchStopReason) =
          matchWithOccupiedPrice(order, uint16(bmi * 256 + bti), maxMatches);
        if (removedLastAtPrice) {
          dbm ^= 2 ** bti;
        }
        if (matchStopReason == MatchStopReason.PriceExhausted) {
          matchStopReason = MatchStopReason.None;
        } else if (matchStopReason != MatchStopReason.None) {
          break;
        }
      }
      bti += 1;
      wbm /= 2;
    }
  }
  // careful - have to do this if broke out of first loop with a match stop reason ...
  if (dbm != cbm) {
    occupiedPriceBitmaps[bmi] = dbm;
  }
  if (matchStopReason == MatchStopReason.None) {
    matchStopReason = MatchStopReason.BookExhausted;
  }
}

// Internal Order Placement.
//
// Match our order against up to maxMatches resting orders at the given price (which
// is known by the caller to have at least one resting order).
//
// The matches (partial or complete) of the resting orders are recorded, and their
// funds are credited.
//
// The order chain for the resting orders is updated, but the occupied price bitmap is NOT -
// the caller must clear the relevant bit if removedLastAtPrice = true is returned.
//
// Only updates the executedBase and executedCntr of our order - caller is responsible
// for e.g. crediting our matched funds, updating status.
//
// Calling with maxMatches == 0 is ok - and expected when the order is a maker-only order.
//
// Returns:
//   removedLastAtPrice:
//     true iff there are no longer any resting orders at this price - caller will need
//     to update the occupied price bitmap.
//
//   matchesLeft:
//     maxMatches passed in minus the number of matches made by this call
```

```
//
//   matchStopReason:
//     If our order is completely matched, matchStopReason will be Satisfied.
//     If our order is not completely matched, matchStopReason will be either:
//        MaxMatches (we are not allowed to match any more times)
//     or:
//        PriceExhausted (nothing left on the book at this exact price)
//
function matchWithOccupiedPrice(
    Order storage ourOrder, uint16 theirPrice, uint maxMatches
  ) internal returns (
  bool removedLastAtPrice, uint matchesLeft, MatchStopReason matchStopReason) {
  matchesLeft = maxMatches;
  uint workingOurExecutedBase = ourOrder.executedBase;
  uint workingOurExecutedCntr = ourOrder.executedCntr;
  uint128 theirOrderId = orderChainForOccupiedPrice[theirPrice].firstOrderId;
  matchStopReason = MatchStopReason.None;
  while (true) {
    if (maxMatches == 0) {
      matchStopReason = MatchStopReason.MaxMatches;
      break;
    }
    uint matchBase;
    uint matchCntr;
    (theirOrderId, matchBase, matchCntr, matchStopReason) =
      matchWithTheirs((ourOrder.sizeBase - workingOurExecutedBase), theirOrderId, theirPrice);
    workingOurExecutedBase += matchBase;
    workingOurExecutedCntr += matchCntr;
    matchesLeft -= 1;
    if (matchStopReason != MatchStopReason.None) {
      break;
    }
  }
  ourOrder.executedBase = workingOurExecutedBase;
  ourOrder.executedCntr = workingOurExecutedCntr;
  if (matchStopReason == MatchStopReason.MaxMatches) {
    removedLastAtPrice = false;
  } else {
    if (theirOrderId == 0) {
      orderChainForOccupiedPrice[theirPrice].firstOrderId = 0;
      orderChainForOccupiedPrice[theirPrice].lastOrderId = 0;
      removedLastAtPrice = true;
    } else {
      orderChainForOccupiedPrice[theirPrice].firstOrderId = theirOrderId;
      orderChainNodeForOpenOrderId[theirOrderId].prevOrderId = 0;
      removedLastAtPrice = false;
    }
  }
}

// Internal Order Placement.
//
// Match up to our remaining amount against a resting order in the book.
//
// The match (partial, complete or effectively-complete) of the resting order
// is recorded, and their funds are credited.
//
// Their order is NOT removed from the book by this call - the caller must do that
```

```
// if the nextTheirOrderId returned is not equal to the theirOrderId passed in.
//
// Returns:
//
//   nextTheirOrderId:
//     If we did not completely match their order, will be same as theirOrderId.
//     If we completely matched their order, will be orderId of next order at the
//     same price - or zero if this was the last order and we've now filled it.
//
//   matchStopReason:
//     If our order is completely matched, matchStopReason will be Satisfied.
//     If our order is not completely matched, matchStopReason will be either
//     PriceExhausted (if nothing left at this exact price) or None (if can continue).
//
function matchWithTheirs(
  uint ourRemainingBase, uint128 theirOrderId, uint16 theirPrice) internal returns (
  uint128 nextTheirOrderId, uint matchBase, uint matchCntr, MatchStopReason matchStopReason) {
  Order theirOrder = orderForOrderId[theirOrderId];
  uint theirRemainingBase = theirOrder.sizeBase - theirOrder.executedBase;
  if (ourRemainingBase < theirRemainingBase) {
    matchBase = ourRemainingBase;
  } else {
    matchBase = theirRemainingBase;
  }
  matchCntr = computeCntrAmountUsingPacked(matchBase, theirPrice);
  if ((ourRemainingBase - matchBase) < baseMinRemainingSize) {
    matchStopReason = MatchStopReason.Satisfied;
  } else {
    matchStopReason = MatchStopReason.None;
  }
  bool theirsDead = recordTheirMatch(theirOrder, theirOrderId, theirPrice, matchBase, matchCntr);
  if (theirsDead) {
    nextTheirOrderId = orderChainNodeForOpenOrderId[theirOrderId].nextOrderId;
    if (matchStopReason == MatchStopReason.None && nextTheirOrderId == 0) {
      matchStopReason = MatchStopReason.PriceExhausted;
    }
  } else {
    nextTheirOrderId = theirOrderId;
  }
}

// Internal Order Placement.
//
// Record match (partial or complete) of resting order, and credit them their funds.
//
// If their order is completely matched, the order is marked as done,
// and "theirsDead" is returned as true.
//
// The order is NOT removed from the book by this call - the caller
// must do that if theirsDead is true.
//
// No sanity checks are made - the caller must be sure the order is
// not already done and has sufficient remaining.
//
function recordTheirMatch(
    Order storage theirOrder, uint128 theirOrderId, uint16 theirPrice, uint matchBase, uint matchCntr
  ) internal returns (bool theirsDead) {
  // they are a maker so no fees
```

```
  theirOrder.executedBase += matchBase;
  theirOrder.executedCntr += matchCntr;
  if (isBuyPrice(theirPrice)) {
    // they have bought base (using the counter they already paid when creating the order)
    balanceBaseForClient[theirOrder.client] += matchBase;
  } else {
    // they have bought counter (using the base they already paid when creating the order)
    balanceCntrForClient[theirOrder.client] += matchCntr;
  }
  if (theirOrder.executedBase >= theirOrder.sizeBase - baseMinRemainingSize) {
    refundUnmatchedAndFinish(theirOrderId, Status.Done, ReasonCode.None);
    MarketOrderEvent(theirOrderId, MarketOrderEventType.CompleteFill, theirPrice, matchBase);
    return true;
  } else {
    MarketOrderEvent(theirOrderId, MarketOrderEventType.PartialFill, theirPrice, matchBase);
    return false;
  }
}

// Internal Order Placement.
//
// Refund any unmatched funds in an order (based on executed vs size) and move to a final state.
//
// The order is NOT removed from the book by this call.
//
// No sanity checks are made - the caller must be sure the order has not already been refunded.
//
function refundUnmatchedAndFinish(uint128 orderId, Status status, ReasonCode reasonCode) internal {
  Order order = orderForOrderId[orderId];
  uint16 price = order.price;
  if (isBuyPrice(price)) {
    uint sizeCntr = computeCntrAmountUsingPacked(order.sizeBase, price);
    balanceCntrForClient[order.client] += sizeCntr - order.executedCntr;
  } else {
    balanceBaseForClient[order.client] += order.sizeBase - order.executedBase;
  }
  order.status = status;
  order.reasonCode = reasonCode;
}

// Internal Order Placement.
//
// Enter a not completely matched order into the book, marking the order as open.
//
// This updates the occupied price bitmap and chain.
//
// No sanity checks are made - the caller must be sure the order
// has some unmatched amount and has been paid for!
//
function enterOrder(uint128 orderId) internal {
  Order order = orderForOrderId[orderId];
  uint16 price = order.price;
  OrderChain orderChain = orderChainForOccupiedPrice[price];
  OrderChainNode orderChainNode = orderChainNodeForOpenOrderId[orderId];
  if (orderChain.firstOrderId == 0) {
    orderChain.firstOrderId = orderId;
    orderChain.lastOrderId = orderId;
    orderChainNode.nextOrderId = 0;
```

```
    orderChainNode.prevOrderId = 0;
    uint bitmapIndex = price / 256;
    uint bitIndex = price % 256;
    occupiedPriceBitmaps[bitmapIndex] |= (2 ** bitIndex);
  } else {
    uint128 existingLastOrderId = orderChain.lastOrderId;
    OrderChainNode existingLastOrderChainNode = orderChainNodeForOpenOrderId[existingLastOrderId];
    orderChainNode.nextOrderId = 0;
    orderChainNode.prevOrderId = existingLastOrderId;
    existingLastOrderChainNode.nextOrderId = orderId;
    orderChain.lastOrderId = orderId;
  }
  MarketOrderEvent(orderId, MarketOrderEventType.Add, price, order.sizeBase - order.executedBase);
  order.status = Status.Open;
}

// Charge the client for the cost of placing an order in the given direction.
//
// Return true if successful, false otherwise.
//
function debitFunds(
    address client, Direction direction, uint sizeBase, uint sizeCntr
  ) internal returns (bool success) {
  if (direction == Direction.Buy) {
    uint availableCntr = balanceCntrForClient[client];
    if (availableCntr < sizeCntr) {
      return false;
    }
    balanceCntrForClient[client] = availableCntr - sizeCntr;
    return true;
  } else if (direction == Direction.Sell) {
    uint availableBase = balanceBaseForClient[client];
    if (availableBase < sizeBase) {
      return false;
    }
    balanceBaseForClient[client] = availableBase - sizeBase;
    return true;
  } else {
    return false;
  }
}

// Public Book View
//
// Intended for public book depth enumeration from web3 (or similar).
//
// Not suitable for use from a smart contract transaction - gas usage
// could be very high if we have many orders at the same price.
//
// Start at the given inclusive price (and side) and walk down the book
// (getting less aggressive) until we find some open orders or reach the
// least aggressive price.
//
// Returns the price where we found the order(s), the depth at that price
// (zero if none found), order count there, and the current blockNumber.
//
// (The blockNumber is handy if you're taking a snapshot which you intend
//  to keep up-to-date with the market order events).
```

```
//
// To walk the book, the caller should start by calling walkBook with the
// most aggressive buy price. If the price returned is the least aggressive
// buy price, the side is complete. Otherwise, call walkBook again with the
// price returned + 1. Then repeat for the sell side.
//
function walkBook(uint16 fromPrice) public constant returns (
    uint16 price, uint depthBase, uint orderCount, uint blockNumber
  ) {
  uint priceStart = fromPrice;
  uint priceEnd = (isBuyPrice(fromPrice)) ? minBuyPrice : maxSellPrice;

  // See comments in matchAgainstBook re: how these crazy loops work.

  uint bmi = priceStart / 256;
  uint bti = priceStart % 256;
  uint bmiEnd = priceEnd / 256;
  uint btiEnd = priceEnd % 256;

  uint wbm = occupiedPriceBitmaps[bmi] >> bti;

  while (bmi < bmiEnd) {
    if (wbm == 0 || bti == 256) {
      bti = 0;
      bmi++;
      wbm = occupiedPriceBitmaps[bmi];
    } else {
      if ((wbm & 1) != 0) {
        // careful - copy-pasted in below loop
        price = uint16(bmi * 256 + bti);
        (depthBase, orderCount) = sumDepth(orderChainForOccupiedPrice[price].firstOrderId);
        return (price, depthBase, orderCount, block.number);
      }
      bti += 1;
      wbm /= 2;
    }
  }
  while (bti <= btiEnd && wbm != 0) {
    if ((wbm & 1) != 0) {
      // careful - copy-pasted in above loop
      price = uint16(bmi * 256 + bti);
      (depthBase, orderCount) = sumDepth(orderChainForOccupiedPrice[price].firstOrderId);
      return (price, depthBase, orderCount, block.number);
    }
    bti += 1;
    wbm /= 2;
  }
  return (uint16(priceEnd), 0, 0, block.number);
}

// Internal Book View.
//
// See walkBook - adds up open depth at a price starting from an
// order which is assumed to be open. Careful - unlimited gas use.
//
function sumDepth(uint128 orderId) internal constant returns (uint depth, uint orderCount) {
  while (true) {
    Order order = orderForOrderId[orderId];
```

```
      depth += order.sizeBase - order.executedBase;
      orderCount++;
      orderId = orderChainNodeForOpenOrderId[orderId].nextOrderId;
      if (orderId == 0) {
        return (depth, orderCount);
      }
    }
  }
}
```