```
// TxSim.cpp - TRANSACTION PROCESSING SYSTEM SIMULATOR
//
// MODULE INDEX
// NAME                          CONTENTS
// Scheduler_c::Cycle            Execute the next operation
// Process_c::SleepUntil         Sleep until a virtual time
// Process_c::Init               Initiate a process
// Client_c::Wakeup              Send the next request
// Client_c::Init                Initialise a client
// Client_c::ProcResponse        Process a response from the master
// SingleBufSlave_c::Wakeup      Wakeup after processing a transaction on the
//                               slave computer
// SingleBufSlave_c::Init        Initialise a single-buffered slave
// SingleBufSlave_c::ProcBatch   Process a batch of transactions received from
//                               the master
// DoubleBufSlave_c::Wakeup      Wakeup after processing a transaction on the
//                               slave computer
// DoubleBufSlave_c::Init        Initialise a double-buffered slave
// DoubleBufSlave_c::ProcBatch   Process a batch of transactions received from
//                               the master
// Master_c::Wakeup              Signal the end of processing of a request
// Master_c::ProcessReq          Process the next request in the request queue
// Master_c::SendTransBatch      Send a transfer batch to the slave
// Master_c::Init                Initialise the master transaction processor
// Master_c::SubmitReq           Process the submission of a request
// Simulator_c::Wakeup           Signal the end of a simulation run
// Simulator_c::Init             Initialise the simulator
// Simulator_c::Simulate         Simulate until a specified time
// SlaveLagIncreasing            Test whether slave lag is increasing
// DetermineLoad                 Determine threshold load by trial and error
// main                          Main line
//
// MAINTENANCE HISTORY
// DATE         PROGRAMMER AND DETAILS
// 19-09-13     JS      Original
// 02-10-13     FYL     Removed unused variables
//
//-----------------------------------------------------------------------------

#include <cstdlib>              // C-style standard library
#include <cstring>              // C-style string manipulation functions
#include <iostream>             // C++ I/O stream declarations
#include <list>                 // C++ list declarations
using namespace std;            // Expand the standard namespace

//-----------------------------------------------------------------------------

// DEFINITIONS

const double    MIN_LOAD = 10.0;
                                // Minimum trial-and-error load
const double    MAX_LOAD = 10000.0;
                                // Maximum trial-and-error load
const size_t    TRIAL_AND_ERROR_ITERATIONS = 12;
                                // Number of trial-and-error iterations
const size_t    CLIENT_CNT = 8;
                                // Number of ordinary clients
const double    INTER_SAMPLE_TIME = 60.0 * 5;

//-----------------------------------------------------------------------------
```

```
// SCHEDULER CLASS

class Scheduler_c {

        // Process Queue

        typedef list<class Process_c*> ProcessList_t;
                                // Process list type definition
        typedef ProcessList_t::iterator ProcessIter_t;
                                // Process iterator type definition

        // Private Variables

        ProcessList_t   processList;
                                // Process list
        double          currentTime;
                                // Current virtual time
        // Public Methods
public:
        void Init () { currentTime = 0; }
                                // Initialise the scheduler
        void Cycle ();
                                // Execute the next operation
        friend class Process_c;
                                // Processes can access private data
};

//---------------------------------------------------------------------------

// PROCESS SUPER-CLASS

class Process_c {
        Scheduler_c     *scheduler;
                                // Pointer to scheduler instance
        double          wakeupTime;
                                // Wakeup time
        bool            sleeping;
                                // Sleeping (versus running)
protected:
        double GetTime () { return scheduler->currentTime; }
                                // Get the current virtual time
        void SleepUntil (double wakeupTime);
                                // Sleep until a specific virtual time
public:
        void Init (Scheduler_c *scheduler);
                                // Initialise
        virtual void Wakeup () = 0;
                                // Wakeup after sleeping
        friend class Scheduler_c;
                                // Schedulers are friends
};

//---------------------------------------------------------------------------

// REQUEST TYPE ENUMERATOR

enum ReqType_t {
        REQ_ORDINARY,           // Ordinary request
        REQ_TRANSFER            // Transfer request
};
```

```
//---------------------------------------------------------------------------

// REQUEST STRUCTURE

struct Req_t {
        ReqType_t       reqType;        // Request type
        class Client_c  *client;        // Pointer to client instance
        double          procTime;       // Request processing time
        size_t          recSize;        // Record size
};
typedef list<Req_t> ReqList_t;
                                // Request list
typedef ReqList_t::iterator ReqIter_t;
                                // Request iterator

//---------------------------------------------------------------------------

// CLIENT CLASS

class Client_c : private Process_c {

        // Private Variables

        class Master_c  *master;
                                // Pointer to master instance
        double          load;
                                // Current load
        double          nextReqTime;
                                // Next request time
        // Private Methods

        void Wakeup ();
                                // Wakeup after sleeping
        // Public Methods
public:
        void Init (Scheduler_c *scheduler, class Master_c *master, double load);
                                // Initialise the client
        void ProcResponse ();
                                // Process a response
};

//---------------------------------------------------------------------------

// BASIC SLAVE

class BasicSlave_c : public Process_c {

        // Public Methods
public:
        virtual void Init (Scheduler_c *scheduler, class Master_c *master) = 0;
                                // Initialise the slave
        virtual void ProcBatch (ReqList_t *reqList) = 0;
                                // Process a batch of transactions
        virtual long GetSlaveTxCnt () = 0;
                                // Get the slave transaction count
};

//---------------------------------------------------------------------------

// SINGLE-BUFFERED SLAVE
```

```
class SingleBufSlave_c : public BasicSlave_c {

        // Private Variables

        class Master_c  *master;
                                // Pointer to master instance
        ReqList_t       transBatch;
                                // Current transfer batch
        long            slaveTxCnt;
                                // Slave transaction count
        // Private Methods

        void Wakeup ();
                                // Wakeup after sleeping
        // Public Methods
public:
        void Init (Scheduler_c *scheduler, class Master_c *master);
                                // Initialise the slave
        void ProcBatch (ReqList_t *transBatch);
                                // Process a batch of transactions
        long GetSlaveTxCnt () { return slaveTxCnt; }
                                // Get the slave transaction count
};

//---------------------------------------------------------------------------

// DOUBLE-BUFFERED SLAVE

class DoubleBufSlave_c : public BasicSlave_c {

        // Double-Buffered Slave State

        enum DoubleState_t {
                DOUBLE_RECV,            // Receiving a batch
                DOUBLE_PROC_RECV,       // Proc a batch and receiving another
                DOUBLE_PROC_READY       // Proc a batch and another is ready
        };

        // Private Variables

        class Master_c  *master;
                                // Pointer to master instance
        DoubleState_t   doubleState;
                                // Slave state
        ReqList_t       transBatch;
                                // Current transfer batch
        ReqList_t       nextBatch;
                                // Next transfer batch
        long            slaveTxCnt;
                                // Slave transaction count
        // Private Methods

        void Wakeup ();
                                // Wakeup after sleeping
        // Public Methods
public:
        void Init (Scheduler_c *scheduler, class Master_c *master);
                                // Initialise the slave
        void ProcBatch (ReqList_t *transBatch);
                                // Process a batch of transactions
```

```
        long GetSlaveTxCnt () { return slaveTxCnt; }
                                // Get the slave transaction count
};

//---------------------------------------------------------------------------

// MASTER TRANSACTION PROCESSOR CLASS

class Master_c : private Process_c {

        // State Enumerator

        enum MasterState_t {
                MASTER_IDLE,            // Master is idle
                MASTER_PROCESSING       // Master is processing a request
        };

        // Private Variables

        MasterState_t   masterState;
                                // Current master state
        BasicSlave_c    *slave;
                                // Pointer to slave instance
        size_t          transBatchCap;
                                // Transfer batch capacity
        long            masterTxCnt;
                                // Master transaction count
        ReqList_t       txQueue;
                                // Transaction queue
        ReqList_t       slaveQueue;
                                // Transactions queued to the slave
        bool            slaveReady;
                                // Slave ready flag
        // Private Methods

        void Wakeup ();
                                // Wakeup after sleeping
        void ProcessReq ();
                                // Process the next request
        void SendTransBatch ();
                                // Send transfer batch
        // Public Methods
public:
        void Init (Scheduler_c *scheduler, BasicSlave_c *slave,
                size_t transBatchCap);
                                // Initialise the master
        void SubmitReq (const Req_t *req);
                                // Submit a request
        long GetMasterTxCnt () { return masterTxCnt; }
                                // Get the master transaction count
};

//---------------------------------------------------------------------------

// SIMULATOR SUPER-CLASS

class Simulator_c : private Process_c {
        bool            endOfRun;
                                // End of simulation run flag
protected:
        Scheduler_c     scheduler;
```

```
                                // Scheduler instance
        Client_c        clientArr[CLIENT_CNT];
                                // Client process array
        Master_c        master;
                                // Master instance
        BasicSlave_c    *slave;
                                // Pointer to slave instance
        // Private Methods
private:
        void Wakeup ();
                                // Wakeup after sleeping
        // Public Methods
public:
        void Init (double load, BasicSlave_c *slave, size_t transBatchCap);
                                // Initialise the simulator
        void Simulate (double wakeupTime);
                                // Simulate until a specified time
        long GetSlaveLag ()
                { return master.GetMasterTxCnt() - slave->GetSlaveTxCnt(); }
                                // Get the current slave lag
};

//------------------------------------------------------------------------

// EXECUTE THE NEXT OPERATION

void
Scheduler_c::Cycle ()
{
        Process_c       *process;       // The next process

        if (processList.begin() == processList.end()) {
                cerr << "Error: empty process list\n";
                abort ();
        }
        process = processList.front();
        if ( ! process->sleeping) {
                cerr << "Error: Cycle: not sleeping\n";
                abort ();
        }
        processList.pop_front();
        currentTime = process->wakeupTime;
        process->sleeping = 0;
        process->Wakeup ();
}

//------------------------------------------------------------------------

// SLEEP UNTIL A VIRTUAL TIME

void
Process_c::SleepUntil (
        double          xWakeupTime)
{
        Scheduler_c::ProcessIter_t processIter;
                                // Process iterator

        if (sleeping) {
                cerr << "Error: SleepUntil: sleeping\n";
                abort ();
        }
```

```
        wakeupTime = xWakeupTime;
        processIter = scheduler->processList.begin();
        while (
                processIter != scheduler->processList.end() &&
                (*processIter)->wakeupTime <= xWakeupTime
        ) processIter ++ ;
        scheduler->processList.insert (processIter, this);
        sleeping = 1;
}

//--------------------------------------------------------------------------

// INITIATE A PROCESS

void
Process_c::Init (
        Scheduler_c     *xScheduler)
{
        scheduler = xScheduler;
        sleeping = 0;
}

//--------------------------------------------------------------------------

// SEND THE NEXT REQUEST

void
Client_c::Wakeup ()
{
        Req_t           req;            // Request structure
        double          r;              // A pseudo-random value

        // Load the request structure

        req.reqType = REQ_ORDINARY;
        req.client = this;

        // Load the request parameters according to the request size

        r = drand48();
        if (r < 0.75) {
                req.procTime = 0.0005 + drand48() * 0.001;
                req.recSize = 50 + lrand48() % 101;
        }
        else if (r < 0.95) {
                req.procTime = 0.01 + drand48() * 0.01;
                req.recSize = 200 + lrand48() % 301;
        }
        else {
                req.procTime = 0.1 + drand48() * 0.15;
                req.recSize = 1000 + lrand48() % 23001;
        }

        // Load the next request time

        nextReqTime = GetTime() + (4.0 + drand48()*8.0) / load;

        // Submit the request

        master->SubmitReq (&req);
}
```

```
//---------------------------------------------------------------------------

// INITIALISE A CLIENT

void
Client_c::Init (
        Scheduler_c     *scheduler,     // Pointer to scheduler instance
        Master_c        *xMaster,       // Pointer to master instance
        double          xLoad)          // Transaction load
{
        // Initialise the class instance variables

        master = xMaster;
        load = xLoad;

        // Initialise the process super-class

        Process_c::Init (scheduler);

        // Schedule the first request

        nextReqTime = GetTime() + (4.0 + drand48()*8.0) / load;
        SleepUntil (nextReqTime);
}

//---------------------------------------------------------------------------

// PROCESS A RESPONSE FROM THE MASTER

void
Client_c::ProcResponse ()
{
        // If the inter-request time has already expired, send
        // the next request immediately.  Otherwise, wait for
        // the inter-request time to expire

        if (nextReqTime > GetTime())
                SleepUntil (nextReqTime);
        else
                Wakeup ();
}

//---------------------------------------------------------------------------

// WAKEUP AFTER PROCESSING A TRANSACTION ON THE SLAVE COMPUTER

void
SingleBufSlave_c::Wakeup ()
{
        Req_t           req;            // Request structure

        // Increment the slave transaction count

        slaveTxCnt ++ ;

        // Pop the transaction off the transfer batch

        transBatch.pop_front ();

        // If there is another transaction in the transfer batch,
```

```
        // simulate processing of that transaction

        if (transBatch.begin() != transBatch.end()) {
                SleepUntil (GetTime() + transBatch.front().procTime);
        }

        // If there are not more transactions waiting in the transfer
        // batch, request another batch of transactions from
        // the master

        else {
                req.reqType = REQ_TRANSFER;
                req.client = 0;
                req.procTime = 0;
                req.recSize = 0;
                master->SubmitReq (&req);
        }
}

//------------------------------------------------------------------------

// INITIALISE A SINGLE-BUFFERED SLAVE

void
SingleBufSlave_c::Init (
        Scheduler_c     *scheduler,     // Pointer to scheduler instance
        Master_c        *xMaster)       // Pointer to master instance
{
        Req_t           req;            // Request structure

        // Initialise the class instance variables

        master = xMaster;
        slaveTxCnt = 0;

        // Initialise the process super-class

        Process_c::Init (scheduler);

        // Request a transfer batch from the master

        req.reqType = REQ_TRANSFER;
        req.client = 0;
        req.procTime = 0;
        req.recSize = 0;
        master->SubmitReq (&req);
}

//------------------------------------------------------------------------

// PROCESS A BATCH OF TRANSACTIONS RECEIVED FROM THE MASTER

void
SingleBufSlave_c::ProcBatch (
        ReqList_t       *xTransBatch)   // Transfer batch
{
        // Save the transfer batch in class instance storage

        transBatch = *xTransBatch;

        // If the transfer batch is empty, something is wrong
```

```
        if (transBatch.begin() == transBatch.end()) {
                cerr << "Error: SingleBufSlave_c::ProcBatch: empty\n";
                abort ();
        }

        // Simulate processing the first transaction in the batch

        SleepUntil (GetTime() + transBatch.front().procTime);
}

//----------------------------------------------------------------------------

// WAKEUP AFTER PROCESSING A TRANSACTION ON THE SLAVE COMPUTER

void
DoubleBufSlave_c::Wakeup ()
{
        Req_t           req;            // Request structure

        // Increment the slave transaction count

        slaveTxCnt ++ ;

        // Pop the transaction off the transfer batch

        transBatch.pop_front ();

        // If there is another transaction in the transfer batch,
        // simulate processing of that transaction

        if (transBatch.begin() != transBatch.end()) {
                SleepUntil (GetTime() + transBatch.front().procTime);
        }

        // If there are no more transactions waiting in the transfer
        // batch, wait for or start processing the next batch

        else {
                switch (doubleState) {
                case DOUBLE_PROC_RECV:
                        doubleState = DOUBLE_RECV;
                        break;
                case DOUBLE_PROC_READY:
                        doubleState = DOUBLE_PROC_RECV;
                        transBatch = nextBatch;
                        SleepUntil (GetTime() + transBatch.front().procTime);
                        req.reqType = REQ_TRANSFER;
                        req.client = 0;
                        req.procTime = 0;
                        req.recSize = 0;
                        master->SubmitReq (&req);
                        break;
                default:
                        cerr << "DoubleBufSlave_c::Wakeup: bad state "
                                << static_cast<int>(doubleState) << '\n';
                        abort ();
                }
        }
}
```

```
//--------------------------------------------------------------------------

// INITIALISE A DOUBLE-BUFFERED SLAVE

void
DoubleBufSlave_c::Init (
        Scheduler_c     *scheduler,     // Pointer to scheduler instance
        Master_c        *xMaster)       // Pointer to master instance
{
        Req_t           req;            // Request structure

        // Initialise the class instance variables

        master = xMaster;
        slaveTxCnt = 0;
        doubleState = DOUBLE_RECV;

        // Initialise the process super-class

        Process_c::Init (scheduler);

        // Request a transfer batch from the master

        req.reqType = REQ_TRANSFER;
        req.client = 0;
        req.procTime = 0;
        req.recSize = 0;
        master->SubmitReq (&req);
}

//--------------------------------------------------------------------------

// PROCESS A BATCH OF TRANSACTIONS RECEIVED FROM THE MASTER

void
DoubleBufSlave_c::ProcBatch (
        ReqList_t       *xTransBatch)   // Transfer batch
{
        Req_t           req;            // Request structure

        // If the transfer batch is empty, something is wrong

        if (xTransBatch->begin() == xTransBatch->end()) {
                cerr << "Error: DoubleBufSlave_c::ProcBatch: empty\n";
                abort ();
        }

        // Process the received transfer batch according to the current state

        switch (doubleState) {

        // Receiving a transfer batch

        case DOUBLE_RECV:

                // Save the transfer batch

                transBatch = *xTransBatch;

                // Update the state
```

```
                doubleState = DOUBLE_PROC_RECV;

                // Initiate processing of the first transaction in the batch

                SleepUntil (GetTime() + transBatch.front().procTime);

                // Request another transfer batch in parallel
                // with processing this transfer batch

                req.reqType = REQ_TRANSFER;
                req.client = 0;
                req.procTime = 0;
                req.recSize = 0;
                master->SubmitReq (&req);
                break;

        // Processing a request while receiving another transfer batch

        case DOUBLE_PROC_RECV:

                // Save the received transfer batch

                nextBatch = *xTransBatch;

                // Update the state to indicate that the next transfer
                // batch is ready

                doubleState = DOUBLE_PROC_READY;
                break;

        // Other states are weird

        default:
                cerr << "DoubleBufSlave_c::Wakeup: bad state "
                        << static_cast<int>(doubleState) << '\n';
                abort ();
        }
}

//----------------------------------------------------------------------------

// SIGNAL THE END OF PROCESSING OF A REQUEST

void
Master_c::Wakeup ()
{
        // Increment the master transaction count

        masterTxCnt ++ ;

        // Send the response to the client

        txQueue.front().client->ProcResponse ();

        // Transfer the transaction to the slave queue

        slaveQueue.push_back (txQueue.front());
        txQueue.pop_front ();

        // If the slave is waiting for a transfer batch,
        // send the transaction to the slave
```

```
        if (slaveReady) SendTransBatch ();

        // If there is another request waiting in the request
        // queue, start processing it, otherwise revert to the idle state

        if (txQueue.begin() != txQueue.end())
                ProcessReq ();
        else
                masterState = MASTER_IDLE;
}

//----------------------------------------------------------------------------

// PROCESS THE NEXT REQUEST IN THE REQUEST QUEUE

void
Master_c::ProcessReq ()
{
        // Process the different request types

        switch (txQueue.front().reqType) {

        // Ordinary requests

        case REQ_ORDINARY:
                masterState = MASTER_PROCESSING;
                SleepUntil (GetTime() + txQueue.front().procTime);
                break;

        // Transfer requests

        case REQ_TRANSFER:
                txQueue.pop_front ();
                slaveReady = 1;
                if (slaveQueue.begin() != slaveQueue.end())
                        SendTransBatch ();
                if (txQueue.begin () != txQueue.end())
                        ProcessReq ();
                else
                        masterState = MASTER_IDLE;
                break;

        // Other request types are weird

        default:
                cerr << "Error: bad request type "
                        << static_cast<int>(txQueue.front().reqType) << '\n';
                abort ();
        }
}

//----------------------------------------------------------------------------

// SEND A TRANSFER BATCH TO THE SLAVE

void
Master_c::SendTransBatch ()
{
        ReqList_t       transBatch;     // Transfer batch
        size_t          batchSize;      // Current batch size
```

```
        // Copy requests in the slave queue to the transfer
        // batch until the slave queue is empty or the transfer
        // batch is full

        batchSize = 0;
        while (
                slaveQueue.begin() != slaveQueue.end() &&
                batchSize + slaveQueue.front().recSize < transBatchCap
        ) {
                batchSize += slaveQueue.front().recSize;
                transBatch.push_back (slaveQueue.front());
                slaveQueue.pop_front ();
        }

        // Reset the slave ready flag

        slaveReady = 0;

        // Send the transfer batch to the slave

        slave->ProcBatch (&transBatch);
}

//------------------------------------------------------------------------------

// INITIALISE THE MASTER TRANSACTION PROCESSOR

void
Master_c::Init (
        Scheduler_c     *scheduler,     // Pointer to scheduler instance
        BasicSlave_c    *xSlave,        // Pointer to slave instance
        size_t          xTransBatchCap) // Transfer batch capacity
{
        // Initialise the class instance variables

        masterState = MASTER_IDLE;
        slave = xSlave;
        transBatchCap = xTransBatchCap;
        masterTxCnt = 0;
        txQueue.clear ();
        slaveQueue.clear ();
        slaveReady = 0;

        // Initialise the process super-class

        Process_c::Init (scheduler);
}

//------------------------------------------------------------------------------

// PROCESS THE SUBMISSION OF A REQUEST

void
Master_c::SubmitReq (
        const Req_t     *req)           // Pointer to request
{
        // Append the request to the transaction queue

        txQueue.push_back (*req);
```

```
        // If the Master Transaction Processor is idle, initiate
        // processing of the request forthwith

        if (masterState == MASTER_IDLE)
                ProcessReq ();
}

//---------------------------------------------------------------------------

// SIGNAL THE END OF A SIMULATION RUN

void
Simulator_c::Wakeup ()
{
        endOfRun = 1;
}

//---------------------------------------------------------------------------

// INITIALISE THE SIMULATOR

void
Simulator_c::Init (
        double          load,           // Transaction load
        BasicSlave_c    *xSlave,        // Pointer to slave instance
        size_t          transBatchCap)  // Transaction batch capacity
{
        size_t          i;              // General purpose index

        // Load the class instance variables

        slave = xSlave;

        // Initialise the scheduler

        scheduler.Init ();

        // Initialise the master

        master.Init (&scheduler, slave, transBatchCap);

        // Initialise each client

        for (i = 0; i < CLIENT_CNT; i++)
                clientArr[i].Init (&scheduler, &master, load);

        // Initialise the slave

        slave->Init (&scheduler, &master);

        // Initialise the shutdown process

        Process_c::Init (&scheduler);
}

//---------------------------------------------------------------------------

// SIMULATE UNTIL A SPECIFIED TIME

void
Simulator_c::Simulate (
```

```
        double          wakeupTime)     // Wakeup time
{
        endOfRun = 0;
        SleepUntil (wakeupTime);
        while ( ! endOfRun) scheduler.Cycle ();
}

//---------------------------------------------------------------------------

// TEST WHETHER SLAVE LAG IS INCREASING

bool
SlaveLagIncreasing (
        double          load,           // Transaction load
        BasicSlave_c    *slave,         // Pointer to slave instance
        size_t          transBatchCap)  // Transaction batch capacity
{
        Simulator_c     simulator;      // Simulator instance
        long            prevSlaveLag;   // Previous slave lag
        size_t          i;              // General purpose index

        // Initialise the simulator

        simulator.Init (load, slave, transBatchCap);

        // Simulate for one minute

        simulator.Simulate (INTER_SAMPLE_TIME);
        for (i = 0; i < 3; i++) {
                prevSlaveLag = simulator.GetSlaveLag();
                simulator.Simulate ((i+2)*INTER_SAMPLE_TIME);
                if (simulator.GetSlaveLag() <= prevSlaveLag) return 0;
        }
        return 1;
}

//---------------------------------------------------------------------------

// DETERMINE THRESHOLD LOAD BY TRIAL AND ERROR

double
DetermineLoad (
        BasicSlave_c    *slave,         // Pointer to slave instance
        size_t          transBatchCap)  // Transaction batch capacity
{
        double          load;           // Tested load
        double          minLoad;        // Minimum load
        double          maxLoad;        // Maximum load
        size_t          i;              // General purpose index

        minLoad = MIN_LOAD;
        maxLoad = MAX_LOAD;
        for (i = 0; i < TRIAL_AND_ERROR_ITERATIONS; i++) {
                load = (minLoad + maxLoad) / 2;
                if (SlaveLagIncreasing (load, slave, transBatchCap))
                        maxLoad = load;
                else
                        minLoad = load;
        }
        return (minLoad + maxLoad) / 2;
}
```

```
//---------------------------------------------------------------------------

// MAIN LINE

int
main ()
{
        SingleBufSlave_c singleBufSlave;
                                        // Single-buffered slave
        DoubleBufSlave_c doubleBufSlave;
                                        // Double-buffered slave

        // Seed the pseudo-random number generator

        srand48 (20365);

        // Simulate the three situations

        cout << "Current           "
                << DetermineLoad (&singleBufSlave, 32768) << '\n';
        cout << "Double buffering  "
                << DetermineLoad (&doubleBufSlave, 32768) << '\n';
        cout << "Bigger batch      "
                << DetermineLoad (&singleBufSlave, 131072) << '\n';
        return 0;
}
```