Source code - table of contents

3 Source code

3.1 Changes to supplied source code

Where appropriate, changes to the supplied source code are shown as diffs, and summaries of the changes are given. For files where many parts have changed and a diff would be hard to read, the entire new file is shown.

I wrote all the changes to the supplied source code listed here, except for the changes to `logsim.cc` and `logsim.h`, which were performed partly by other team members.

3.1.1 devices

Summary: Three functions added, mainly for use by the device editing part of the GUI. Small modification to `updateclocks()` so that changing `frequency` after running the simulation can no longer cause a clock to stop changing state. `maxmachinecycles` is now dynamically set depending on the number of devices, to make it less likely that simulation will fail unnecessarily. It will still fail if an inverter is connected to itself, but will no longer fail for 20 inverters connected in series.

Listing 1: devices.h diff

```
@@ -55,6 +55,15 @@ class devices {
  void debug (bool on);
  /* Used to set debugging switch.                                    */

+ // Changes the number of inputs for a gate
+ void SetGateInputCount(devlink d, int newCount);
+
+ // Gets the name for an input of a gate (n=1 -> 'I1', n=2 -> 'I2', etc)
+ name GetGateInputName(int n);
+
+ // Checks whether all inputs are connected
+ bool CheckDeviceInputs(devlink d);
+
  devices (names* names_mod, network* net_mod);
  /* Called to initialise module.                                     */
 };
```

Listing 2: devices.cc diff

```
@@ -125,14 +125,7 @@ void devices::makegate (devicekind dkind, name did, int
    ninputs, bool& ok)
    netz->adddevice (dkind, did, d);
    netz->addoutput (d, blankname);
    for (n = 1; n <= ninputs; n++) {
-     iname = "I";
-     if (n < 10) {
-       iname += ((char) n) + '0';
-     } else {
-       iname += ((char) (n / 10)) + '0';
-       iname += ((char) (n % 10)) + '0';
-     }
-     netz->addinput (d, nmz->lookup (iname));
+     netz->addinput(d, GetGateInputName(n));
    }
  }
 }
@@ -342,7 +335,7 @@ void devices::updateclocks (void)
  devlink d;
  for (d = netz->devicelist (); d != NULL; d = d->next) {
    if (d->kind == aclock) {
-     if (d->counter == d->frequency) {
+     if (d->counter >= d->frequency) {
        d->counter = 0;
        if (d->olist->sig == high)
          d->olist->sig = falling;
@@ -364,7 +357,8 @@ void devices::updateclocks (void)
  */
 void devices::executedevices (bool& ok)
 {
- const int maxmachinecycles = 20;
+ int maxmachinecycles = 20;
+ int count = 0;
  devlink d;
  int machinecycle;
  if (debugging)
@@ -387,9 +381,11 @@ void devices::executedevices (bool& ok)
      case xorgate:  execxorgate (d);          break;
      case dtype:    execdtype (d);            break;
      }
+     if (machinecycle==1) count++;
      if (debugging)
        showdevice (d);
    }
+   if (machinecycle==1) maxmachinecycles = 20 + 2*count;
  } while ((! steadystate) && (machinecycle < maxmachinecycles));
  if (debugging)
    cout << "End of execution cycle" << endl;
@@ -464,4 +460,48 @@ void devices::debug (bool on)
  qbarpin = nmz->lookup("QBAR");
 }

+name devices::GetGateInputName(int n)
+{
+ // moved from makegate() into new function
+ namestring iname = "I";
+ if (n < 10) {
+   iname += ((char) n) + '0';
+ } else {
+   iname += ((char) (n / 10)) + '0';
+   iname += ((char) (n % 10)) + '0';
+ }
+ return nmz->lookup(iname);
+}
+
+void devices::SetGateInputCount(devlink d, int newCount)
+{
+ if (!d) return;
+ int oldCount = GetLinkedListLength(d->ilist);
+ if (newCount<oldCount)
+ {
+   for (int i=0; i<oldCount-newCount; i++)
+   {
+     inplink nextI = d->ilist->next;
+     delete d->ilist;
+     d->ilist = nextI;
+   }
+ }
+ else
+ {
+   for (int i=oldCount+1; i<=newCount; i++)
+   {
+     netz->addinput(d, GetGateInputName(i));
+   }
+ }
+}

+bool devices::CheckDeviceInputs(devlink d)
+{
+ inplink i = d->ilist;
+ while (i != NULL)
+ {
```

```
+   if (i->connect == NULL) return false;
+   i = i->next;
+ }
+ return true;
+}
```

3.1.2 network

Summary: Functions added for getting linked list length, deleting or disconnecting a device, and obtaining device and input/output name strings. `outputrec` now has a pointer to the parent device, to make it easier for the device editing part of the GUI to find which device an output is connected to.

Listing 3: network.h diff

```
@@ -2,6 +2,9 @@
 #define network_h

 #include "names.h"
+#include <string>
+
+using namespace std;

 /* Network specification */

@@ -10,10 +13,12 @@
               norgate, xorgate, dtype, baddevice
              } devicekind;

+struct devicerec;
 struct outputrec {
  name       id;
  asignal    sig;
  outputrec* next;
+ devicerec* dev;
 };
 typedef outputrec* outplink;
 struct inputrec {
@@ -36,6 +41,18 @@ struct devicerec {
 };
 typedef devicerec* devlink;

+template <class T>
+int GetLinkedListLength(T item)
+{
+ int count = 0;
+ while (item != NULL)
+ {
+   count++;
+   item = item->next;
+ }
+ return count;
+}
+
 class network {
  names* nmz;  // the instatiation of the names class that we are going to
    use.

@@ -72,12 +89,23 @@ class network {
  /* 'outp' output of device 'odev'. 'ok' is set true if operation       */
  /* succeeds.                                                           */

- void checknetwork (bool& ok);
+ void checknetwork (bool& ok, bool silent=false);
  /* Checks that all inputs are connected to an output.                  */

  network (names* names_mod);
  /* Called on system initialisation.                                     */

+ string getsignalstring(name dev, name p=blankname);
+ string getsignalstring(devlink dev, outplink o);
+ string getsignalstring(devlink dev, inplink i);
+ /* Returns the string corresponding to the given device and pin        */
+
+ // Disconnects all inputs connected to the given output
+ void disconnectoutput(outplink o);
+
+ // Deletes a device (after disconnecting the outputs)
+ void deletedevice(devlink dTarget);
+
 private:
  devlink devs;          // the list of devices
  devlink lastdev;       // last device in list of devices
```

Listing 4: network.cc diff

```
@@ -133,6 +133,7 @@ void network::addinput (devlink dev, name iid)
 void network::addoutput (devlink dev, name oid)
 {
  outplink o = new outputrec;
+ o->dev = dev;
```

```
  o->id = oid;
  o->sig = low;
  o->next = dev->olist;
@@ -170,14 +171,17 @@ void network::makeconnection (name idev, name inp, name
     odev, name outp, bool& o
  * Checks that all inputs are connected to an output.
  *
  */
-void network::checknetwork (bool& ok)
+void network::checknetwork (bool& ok, bool silent)
 {
  devlink d;
  inplink i;
  ok = true;
  for (d = devs; d != NULL; d = d->next)
    for (i = d->ilist; i != NULL; i = i->next)
-     if (i->connect == NULL) {
+     if (i->connect == NULL)
+     {
+       if (!silent)
+       {
        cout << "Unconnected Input : ";
        nmz->writename (d->id);
        if (i->id != blankname) {
@@ -185,6 +189,7 @@ void network::checknetwork (bool& ok)
          nmz->writename (i->id);
        }
        cout << endl;
+       }
        ok = false;
      }
 }
@@ -205,3 +210,87 @@ void network::checknetwork (bool& ok)
  lastdev = NULL;
 }

+// Delete a device, after disconnecting it. Monitors should be deleted too,
    but this function only handles deleting the devicerec and inputs+outputs
     (use circuit::RemoveDevice instead).
+void network::deletedevice(devlink dTarget)
+{
+ if (!dTarget) return;
+ devlink d = devicelist();
+ devlink dPrev = NULL;
+ while (d!=NULL)
+ {
+   if (d == dTarget)
+   {
+     outplink o = d->olist, oNext;
+     while (o != NULL)
+     {
+       disconnectoutput(o);
+       oNext = o->next;
+       delete o;
+       o = oNext;
+     }
+     inplink i = d->ilist, iNext;
+     while (i != NULL)
+     {
+       iNext = i->next;
+       delete i;
+       i = iNext;
+     }
+     if (devs == d)
+       devs = d->next;
+     else
+       dPrev->next = d->next;
+     if (lastdev == d)
+       lastdev = dPrev;
+     break;
+   }
+   dPrev = d;
+   d = d->next;
+ }
+}
+
+// Disconnects all inputs connected to the given output
+void network::disconnectoutput(outplink o)
+{
+ devlink d = devicelist();
+ while (d!=NULL)
+ {
+   inplink i = d->ilist;
+   while (i!=NULL)
+   {
+     if (i->connect == o)
+       i->connect = NULL;
+     i = i->next;
+   }
+   d = d->next;
+ }
+}
```

```
+
+string network::getsignalstring(name dev, name p)
+{
+ if (dev==blankname)
+   return "";
+ string str = nmz->getnamestring(dev);
+ if (p != blankname)
+   str += "." + nmz->getnamestring(p);
+ return str;
+}
+
+string network::getsignalstring(devlink d, outplink o)
+{
+ if (d==NULL)
+   return "";
+ name opn = blankname;
+ if (o==NULL)
+   return getsignalstring(d->id);
+ return getsignalstring(d->id, o->id);
+}
+
+string network::getsignalstring(devlink d, inplink i)
+{
+ if (d==NULL)
+   return "";
+ name opn = blankname;
+ if (i==NULL)
+   return getsignalstring(d->id);
+ return getsignalstring(d->id, i->id);
+}
```

3.1.3 logsim

Summary: Command line argument is now optional for the GUI, and a `MyFrame` member function is used to load the file if using the GUI. Additional class `error` added to count and display errors which occur during parsing.

Listing 5: logsim.h diff

```
@@ -7,6 +7,7 @@
 #include "monitor.h"
 #include "network.h"
 #include "parser.h"
+#include "scanner.h"
 class MyApp: public wxApp
 {
@@ -19,6 +20,8 @@ class MyApp: public wxApp
  monitor *mmz;  // pointer to the monitor class
  scanner *smz;  // pointer to the scanner class
  parser *pmz;   // pointer to the parser class
+ error *erz; // pointer to the error class
+ bool ok;
 };

 #endif /* logsim_h */
```

Listing 6: logsim.cc

```
1  #include "logsim.h"
2  #include "userint.h"
3  #include "gui.h"
4  #include <GL/glut.h>
5
6  #define USE_GUI
7
8  IMPLEMENT_APP(MyApp)
9
10 bool MyApp::OnInit()
11 // This function is automatically called when the application starts
12 {
13 #ifndef USE_GUI
14   if (argc != 2) { // check we have one command line argument
15     wcout << "Usage:      " << argv[0] << " [filename]" << endl;
16     exit(1);
17   }
18 #endif
19
20   // Construct the six classes required by the innards of the logic
       simulator
21   nmz = new names();
22   netz = new network(nmz);
23   dmz = new devices(nmz, netz);
24   mmz = new monitor(nmz, netz);
25
26 #ifdef USE_GUI
27   // glutInit cannot cope with Unicode command line arguments, so we
       pass
28   // it some fake ASCII ones instead
29   char **tmp1; int tmp2 = 0; glutInit(&tmp2, tmp1);
30   // Construct the GUI
31   MyFrame *frame = new MyFrame(NULL, wxT("Logic simulator"),
       wxDefaultPosition,  wxSize(800, 600), nmz, dmz, mmz, netz);
32   frame->Show(true);
```

```
  if (argc == 2)
  {
    frame->loadFile(wxString(argv[1]).mb_str());
  }
  return(true); // enter the GUI event loop
#else
  smz = new scanner(nmz, wxString(argv[1]).mb_str(), ok);
  if (!ok)
  {
    return(false);
  }
  erz = new error(smz);
  pmz = new parser(netz, dmz, mmz, smz, erz);
  if (pmz->readin ()) { // check the logic file parsed correctly
    // Construct the text-based interface
    userint umz(nmz, dmz, mmz);
    umz.userinterface();
  }
  return(false); // exit the application
#endif /* USE_GUI */
}
```

3.1.4 monitors

Summary: Converted to use STL vectors, and to use pointers to the device and output objects instead of the `name` for each. New functions `getsamplecount`, `getsignalstring`, `IsMonitored` added.

Listing 7: monitor.h diff

```
@@ -4,27 +4,28 @@
 #include "names.h"
 #include "network.h"
 #include "devices.h"
+#include <vector>
+#include <string>
+using namespace std;

 const int maxmonitors = 10;      /* max number of monitor points */
 const int maxcycles = 50;        /* max number of cycles per run */

+typedef vector<asignal> signaltrace;
+
 struct moninfo {
- name devid;
+ devlink d;
  outplink op;
+ signaltrace disp;
 };
-struct monitortable {
- int used;
- moninfo sigs[maxmonitors];
-};
-typedef asignal signaltrace[maxcycles];
+typedef vector<moninfo> montable;

 class monitor {
  names*   nmz;     // version of names class to use.
  network* netz;    // version of the network class to use.

- monitortable mtab;                // table of monitored signals
+ montable mtab;                  // table of monitored signals
  int cycles;                       // counts clock cycles
- signaltrace disp[maxmonitors];

 public:
  void makemonitor (name dev, name outp, bool& ok);
@@ -60,6 +61,16 @@ class monitor {

  monitor (names* names_mod, network* network_mod);
  /* Called to initialise the monitor module.                          */
+
+ // Returns the name of the monitored signal as a string
+ string getsignalstring(int m);
+
+ // Returns the number of samples recorded by the n'th monitor
+ int getsamplecount(int m);
+
+ // Returns true if the given output is being monitored
+ bool IsMonitored(outplink o);
+
 };

 #endif /* monitor_h */
```

Listing 8: monitor.cc

```
#include <iostream>
#include "monitor.h"

using namespace std;

```

```
/*
    ************************************************************************

 *
 * Sets a monitor on the 'outp' output of device 'dev' by placing an
 * entry in the monitor table. 'ok' is set true if operation succeeds
    .
 *
 */
void monitor::makemonitor (name dev, name outp, bool& ok)
{
  devlink d;
  outplink o;
  d = netz->finddevice(dev);
  ok = (d != NULL);
  if (ok)
  {
    o = netz->findoutput(d, outp);
    ok = (o != NULL);
    if (ok)
    {
      moninfo mon;
      mon.d = d;
      mon.op = o;
      mtab.push_back(mon);
    }
  }
}


/*
    ************************************************************************

 *
 * Removes the monitor set on the 'outp' output of device 'dev'. 'ok'
     is
 * set true if operation succeeds.
 *
 */
void monitor::remmonitor (name dev, name outp, bool& ok)
{
  bool found;
  found = false;
  for (montable::iterator it=mtab.begin(); it!=mtab.end(); ++it)
  {
    if ((it->d->id == dev) && (it->op->id == outp))
    {
      found = true;
      mtab.erase(it);
      break;
    }
  }
  ok = found;
}


/*
    ************************************************************************

 *
 * Returns number of signals currently monitored.
 *
 */
int monitor::moncount (void)
{
  return mtab.size();
}


/*
    ************************************************************************

 *
 * Returns signal level of n'th monitor point.
 *
 */
asignal monitor::getmonsignal (int n)
{
  return (mtab[n].op->sig);
}


/*
    ************************************************************************

 *
 * Returns name of n'th monitor.
 *
 */
void monitor::getmonname (int n, name& dev, name& outp)
{
  dev = mtab[n].d->id;
  outp = mtab[n].op->id;
}


/*
    ************************************************************************

 *
 * Initialises monitor memory in preparation for a new output
    sequence.
```

```
 *
 */
void monitor::resetmonitor (void)
{
  int n;
  for (n = 0; n < moncount (); n++)
    mtab[n].disp.clear();
  cycles = 0;
}


/*
    ************************************************************************

 *
 * Called every clock cycle to record the state of each monitored
 * signal.
 *
 */
void monitor::recordsignals (void)
{
  int n;
  for (n = 0; n < moncount (); n++)
    mtab[n].disp.push_back(getmonsignal(n));
  cycles++;
}

/*
    ************************************************************************

 *
 * Access recorded signal trace, returns false if invalid monitor
 * or cycle.
 *
 */
bool monitor::getsignaltrace(int m, int c, asignal &s)
{
  if ((c < cycles) && (m < moncount ()) && c<mtab[m].disp.size()) {
    s = mtab[m].disp[c];
    return true;
  }
  return false;
}

/*
    ************************************************************************

 *
 * Get number of recorded cycles for the monitor, returns -1 if
   invalid
 * monitor.
 *
 */
int monitor::getsamplecount(int m)
{
  if (m < moncount ()) {
    return mtab[m].disp.size();
  }
  return -1;
}

/*
    ************************************************************************

 *
 * Displays state of monitored signals.
 *
 */
void monitor::displaysignals (void)
{
  const int margin = 20;
  int n, i;
  name dev, outp;
  int namesize;
  for (n = 0; n < moncount (); n++) {
    getmonname (n, dev, outp);
    namesize = nmz->namelength (dev);
    nmz->writename (dev);
    if (outp != blankname) {
      cout << ".";
      nmz->writename (outp);
      namesize = namesize + nmz->namelength (outp) + 1;
    }
    if ((margin - namesize) > 0) {
      for (i = 0; i < (margin - namesize - 1); i++)
        cout << " ";
      cout << ":";
    }
    for (i = 0; i < cycles; i++)
      switch (mtab[n].disp[i]) {
      case high:    cout << "-"; break;
      case low:     cout << "_"; break;
      case rising:  cout << "/"; break;
      case falling: cout << "\\"; break;
      }
    cout << endl;
  }
}


/*
    ************************************************************************
```

```
 *
 * Called to initialise the monitor module.
 * Remember the names of the shared names and network modules.
 *
 */
monitor::monitor (names* names_mod, network* network_mod)
{
  nmz = names_mod;
  netz = network_mod;
}

// Returns the name of the monitored signal as a string
string monitor::getsignalstring(int m)
{
  if (m<0 || m>=moncount()) return "";
  return netz->getsignalstring(mtab[m].d->id, mtab[m].op->id);
}

bool monitor::IsMonitored(outplink o)
{
  for (int n = 0; n < moncount (); n++)
  {
    if (mtab[n].op == o) return true;
  }
  return false;
}
```

3.2 GUI

I was responsible for the whole of the GUI, except for some contributions by other team members to the `MyFrame::loadFile()` function.

3.2.1 Helper class: circuit

Listing 9: circuit.h

```
#ifndef circuit_h
#define circuit_h

#include "names.h"
#include "network.h"
#include "devices.h"
#include "monitor.h"
#include "observer.h"
#include <vector>
using namespace std;

class circuit;


// Many bits of the GUI involve manipulating (creating, searching, sorting) vectors where each element needs to contain one or more pointers to circuit elements, with an associated string. This class is a unified way of doing that for all circuit elements (inputs, outputs, and devices).
class CircuitElementInfo
{
public:
  devlink d;
  outplink o;
  inplink i;
  string namestr;
  CircuitElementInfo() : d(NULL), o(NULL), i(NULL), namestr("") {}
  CircuitElementInfo(devlink dev, string str="") : d(dev), o(NULL), i(NULL), namestr(str) {}
  CircuitElementInfo(devlink dev, outplink outp, string str="") : d(dev), o(outp), i(NULL), namestr(str) {}
  CircuitElementInfo(outplink outp, string str="") : d(outp->dev), o(outp), i(NULL), namestr(str) {}
  CircuitElementInfo(devlink dev, inplink inp, string str="") : d(dev), o(NULL), i(inp), namestr(str) {}
};

class CircuitElementInfoVector : public vector<CircuitElementInfo>
{
public:
  // push_back() all devices in a linked list of devices, all outputs in a linked list of devices, or all inputs in a linked list of devices
  void push_back_all_devs(devlink d);
  void push_back_all_outputs(devlink d);
  void push_back_all_inputs(devlink d);
  // push_back() all outputs or inputs for a particular device
  void push_back_dev_outputs(devlink d);
  void push_back_dev_inputs(devlink d);
  // Sets namestr for each element to the device or signal name (e.g. S1, or G1.I1, or DT1.CLK)
  void UpdateSignalNames(circuit* c);
private:
  template <class T> void push_back_iolist(devlink d, T item);
};

bool CircuitElementInfo_namestrcmp(const CircuitElementInfo a, const CircuitElementInfo b);// to alphabetically sort a vector<outputinfo>
bool CircuitElementInfo_iconnect_namestrcmp(const CircuitElementInfo a, const CircuitElementInfo b);// to sort a vector<outputinfo> by input connected state then alphabetically by namestr


// Used in the device editing GUI to store a pointer to the currently selected device and notify widgets when it changes. This is here instead of in a gui file because it contains nothing specific to wxWidgets and may be useful elsewhere
class SelectedDevice
{
private:
  devlink selected;
public:
  ObserverSubject changed;
  devlink Get()
  {
    return selected;
  }
  void Set(devlink d)
  {
    if (d!=selected)
    {
      selected = d;
      changed.Trigger();
    }
  }
};


```

```
// This class is a more convenient way of passing around a set of names, monitors, devices, and network modules.
// It also handles observers for changes to the circuit.
// It provides a home for a few functions that are more closely related to the circuit itself than to the GUI or MyFrame, such as running the network
class circuit
{
private:
  // Module pointers
  names *names_mod;
  devices *devices_mod;
  monitor *monitor_mod;
  network *network_mod;
  bool ownModules;
  void AllocModules();
  void DestroyModules();

  int continuedCycles;// how many simulation cycles were completed last time the run or continue button was used
  int totalCycles;// how many simulation cycles have been completed in total

public:
  circuit();
  circuit(names *existing_names_mod, devices *existing_devices_mod, monitor *existing_monitor_mod, network *existing_network_mod);
  virtual ~circuit();
  // Swap module pointers without changing observers
  void SwapModules(circuit& source);
  // clear the circuit, currently done by allocating new modules
  void Clear();
  // Clear recorded samples
  void ResetMonitors();
  // Simulate the logic circuit, optionally clearing monitors beforehand
  bool Simulate(int ncycles, bool resetBefore=true);
  // GetUnmonitoredOutputs returns true if there are unmonitored outputs in the circuit
  //   unmonitoredOutputsRet is an optional pointer to a vector to hold the list of unmonitored outputs
  bool GetUnmonitoredOutputs(CircuitElementInfoVector *unmonitoredOutputsRet=NULL);

  ObserverSubject monitorsChanged;// addition or removal of monitors
  ObserverSubject circuitChanged;// changes to devices, device properties, and connections
  ObserverSubject monitorSamplesChanged;// simulation has been run or continued, or monitor samples have been cleared

  names* nmz() { return names_mod; }
  devices* dmz() { return devices_mod; }
  monitor* mmz() { return monitor_mod; }
  network* netz() { return network_mod; }
  int GetTotalCycles() { return totalCycles; }
  int GetContinuedCycles() { return continuedCycles; }
  static bool IsDeviceNameValid(string devname);
  void RemoveDevice(devlink d);
};

#endif
```

Listing 10: circuit.cc

```
#include "circuit.h"
#include <algorithm>
using namespace std;

void CircuitElementInfoVector::push_back_all_devs(devlink d)
{
  while (d!=NULL)
  {
    push_back(CircuitElementInfo(d));
    d = d->next;
  }
}

void CircuitElementInfoVector::push_back_all_outputs(devlink d)
{
  while (d!=NULL)
  {
    push_back_dev_outputs(d);
    d = d->next;
  }
}

void CircuitElementInfoVector::push_back_all_inputs(devlink d)
{
  while (d!=NULL)
  {
    push_back_dev_inputs(d);
    d = d->next;
  }
}

void CircuitElementInfoVector::push_back_dev_outputs(devlink d)
{
  push_back_iolist(d, d->olist);
}

void CircuitElementInfoVector::push_back_dev_inputs(devlink d)
{
```

```
  push_back_iolist(d, d->ilist);
}

template <class T>
void CircuitElementInfoVector::push_back_iolist(devlink d, T item)
{
  while (item != NULL)
  {
    push_back(CircuitElementInfo(d, item));
    item = item->next;
  }
}

void CircuitElementInfoVector::UpdateSignalNames(circuit* c)
{
  for (CircuitElementInfoVector::iterator it=begin(); it<end(); it++)
  {
    if (!it->d)
    {
      it->namestr = "";
    }
    else
    {
      if (it->i)
        it->namestr = c->netz()->getsignalstring(it->d,it->i);
      else if (it->o)
        it->namestr = c->netz()->getsignalstring(it->d,it->o);
      else
        it->namestr = c->nmz()->getnamestring(it->d->id);
    }
  }
}



circuit::circuit()
{
  AllocModules();
  totalCycles = continuedCycles = 0;
}

circuit::circuit(names *existing_names_mod, devices *
    existing_devices_mod, monitor *existing_monitor_mod, network *
    existing_network_mod)
{
  names_mod = existing_names_mod;
  devices_mod = existing_devices_mod;
  monitor_mod = existing_monitor_mod;
  network_mod = existing_network_mod;
  ownModules = false;
  totalCycles = continuedCycles = 0;
}

circuit::~circuit()
{
  DestroyModules();
}

void circuit::AllocModules()
{
  names_mod = new names();
  network_mod = new network(names_mod);
  devices_mod = new devices(names_mod, network_mod);
  monitor_mod = new monitor(names_mod, network_mod);
  ownModules = true;
}

void circuit::DestroyModules()
{
  if (ownModules)
  {
    delete monitor_mod;
    delete devices_mod;
    delete network_mod;
    delete names_mod;
  }
  monitor_mod = NULL;
  devices_mod = NULL;
  network_mod = NULL;
  names_mod = NULL;
  ownModules = false;
}

void circuit::Clear()
{
  DestroyModules();
  AllocModules();
  totalCycles = continuedCycles = 0;
  circuitChanged.Trigger();
  monitorsChanged.Trigger();
  monitorSamplesChanged.Trigger();
}

void circuit::SwapModules(circuit& source)
{
  swap(names_mod, source.names_mod);
  swap(devices_mod, source.devices_mod);
  swap(monitor_mod, source.monitor_mod);
  swap(network_mod, source.network_mod);
  swap(ownModules, source.ownModules);
}

```

```
void circuit::ResetMonitors()
{
  mmz()->resetmonitor();
  totalCycles = continuedCycles = 0;
  monitorSamplesChanged.Trigger();
}

bool circuit::Simulate(int ncycles, bool resetBefore)
{
  if (resetBefore)
  {
    mmz()->resetmonitor();
    totalCycles = 0;
  }
  continuedCycles = 0;

  // Function to run the network
  bool ok = true;
  int n = ncycles;

  while (n > 0 && ok)
  {
    dmz()->executedevices(ok);
    if (ok)
    {
      n--;
      totalCycles++;
      continuedCycles++;
      mmz()->recordsignals();
    }
    else
    {
      cout << "Error: network is oscillating" << endl;
      ok = false;
    }
  }

  monitorSamplesChanged.Trigger();
  return ok;
}

bool CircuitElementInfo_namestrcmp(const CircuitElementInfo a, const
    CircuitElementInfo b)
{
  return (a.namestr<b.namestr);
}

bool CircuitElementInfo_iconnect_namestrcmp(const CircuitElementInfo
    a, const CircuitElementInfo b)
{
  if (a.i && b.i && (a.i->connect!=NULL)!=(b.i->connect!=NULL))
    return (a.i->connect!=NULL) < (b.i->connect!=NULL);
  return (a.namestr<b.namestr);
}

bool circuit::GetUnmonitoredOutputs(CircuitElementInfoVector *
    unmonitoredOutputsRet)
{
  CircuitElementInfoVector unmonitoredOutputs;

  int monCount = mmz()->moncount();
  devlink d = netz()->devicelist();
  // Loop through devices
  while (d!=NULL)
  {
    // Loop through device outputs
    outplink o = d->olist;
    while (o!=NULL)
    {
      // Check whether this output is currently being monitored
      if (!mmz()->IsMonitored(o))
      {
        unmonitoredOutputs.push_back(CircuitElementInfo(d,o));
      }
      o = o->next;
    }
    d = d->next;
  }

  // If a pointer to a vector was supplied, put the list of
    unmonitored outputs in there
  if (unmonitoredOutputsRet)
  {
    unmonitoredOutputs.UpdateSignalNames(this);
    unmonitoredOutputsRet->swap(unmonitoredOutputs);
  }

  // Return true if there are some unmonitored outputs in the circuit
  return (unmonitoredOutputs.size()>0);
}

bool circuit::IsDeviceNameValid(string devname)
{
  // Checks syntax of a device name string (but not whether a device
    already exists with that name or if it's a reserved word)
  if (!devname.length())
    return false;
  if (!isalpha(devname[0]))
    return false;
  for (string::iterator it=devname.begin(); it<devname.end(); ++it)
  {
    if (!isalpha(*it) && !isdigit(*it) && (*it)!='_') return false;
```

```
  }
  return true;
}

void circuit::RemoveDevice(devlink d)
{
  if (!d) return;

  // Remove monitors first
  outplink o = d->olist;
  bool ok;
  while (o != NULL)
  {
    mmz()->remmonitor(d->id, o->id, ok);
    o = o->next;
  }

  // Disconnect, release memory, and remove from linked list of
    devices
  netz()->deletedevice(d);

  circuitChanged.Trigger();
  monitorsChanged.Trigger();
}
```

3.2.2 Helper class: observers

Listing 11: observer.h

```
#ifndef observer_h
#define observer_h

#include <vector>
#include <iostream>
using namespace std;

// Abstract base class for storing callback functions
class ObserverCallbackBase
{
public:
  virtual void* GetTarget() = 0;
  virtual void Run() = 0;
  virtual ~ObserverCallbackBase() {}
};

// Derived class for storing callbacks, using templates to allow
    member functions in different classes of the form "void SomeClass
    :func()"
// to be called without needing a different derived ObserverCallback
    class for each SomeClass
template <class T>
class ObserverCallback : public ObserverCallbackBase
{
private:
  T* targetObject;//pointer to the object which will have a member
    function called
  void (T::*targetFunc)();//pointer to the member function to call
public:
  ObserverCallback(T* obj, void (T::*func)())
  {
    targetObject = obj;
    targetFunc = func;
  }
  virtual void* GetTarget()
  {
    // Return the pointer to the target object, used in
    ObserverSubject.Detach() to check whether this callback is for
    the object being detached
    return targetObject;
  }
  virtual void Run()
  {
    // Run the callback function
    (targetObject->*targetFunc)();
  }
};

class ObserverSubject
{
private:
  vector<ObserverCallbackBase*> callbacks;
  bool callingFuncs;// Whether Trigger() is currently being run
public:
  ObserverSubject() : callingFuncs(false) {}

  // Register the member function targetFunc of targetObj to be
    called when this ObserverSubject is triggered
  // Does not currently detect and prevent duplicates
  template <class T>
  void Attach(T* targetObj, void (T::*targetFunc)())
  {
    callbacks.push_back(new ObserverCallback<T>(targetObj, targetFunc
    ));
  }

  // Detach all registered callbacks with the object pointer equal to
     targetObj
  void Detach(void* targetObj)
  {
```

```
    for (vector<ObserverCallbackBase*>::iterator it=callbacks.begin()
    ; it<callbacks.end(); ++it)
    {
      if ((*it)->GetTarget()==targetObj)
      {
        delete *it;
        callbacks.erase(it);
      }
    }
  }

  // Runs all registered callback functions
  void Trigger()
  {
    if (callingFuncs)
    {
      // Prevent recursion
      cout << "Error: ObserverSubject::Trigger was called recursively
    " << endl;
      return;
    }
    callingFuncs = true;
    for (vector<ObserverCallbackBase*>::iterator it=callbacks.begin()
    ; it<callbacks.end(); ++it)
    {
      (*it)->Run();
    }
    callingFuncs = false;
  }

  // Return the number of registered callbacks
  int CallbackCount()
  {
    return callbacks.size();
  }

  virtual ~ObserverSubject()
  {
    for (vector<ObserverCallbackBase*>::iterator it=callbacks.begin()
    ; it<callbacks.end(); ++it)
    {
      delete *it;
    }
  }
};

// Example usage in observer-test.cc

#endif
```

3.2.3 Main GUI: dialogs and frame

Listing 12: gui.h

```
#ifndef gui_h
#define gui_h

#include "gui-canvas.h"
#include <wx/wx.h>
#include <wx/spinctrl.h>
#include <wx/textctrl.h>
#include <wx/listbox.h>
#include <wx/scrolwin.h>
#include <wx/panel.h>
#include "names.h"
#include "devices.h"
#include "monitor.h"
#include "network.h"
#include "circuit.h"
#include <vector>
#include <string>

using namespace std;

class MyFrame: public wxFrame
{
public:
  MyFrame(wxWindow *parent, const wxString& title, const wxPoint& pos
    , const wxSize& size,
          names *names_mod = NULL, devices *devices_mod = NULL,
    monitor *monitor_mod = NULL, network *net_mod = NULL,
          long style = wxDEFAULT_FRAME_STYLE); // constructor
  virtual ~MyFrame();
  bool loadFile(const char * filename);// loads the given file,
    returns true if successful
private:
  MyGLCanvas *canvas;                     // GL drawing area widget
    to draw traces
  wxSpinCtrl *spin;                       // control widget to select
     the number of cycles
  wxTextCtrl *outputTextCtrl;             // textbox to display
    messages sent to cout (e.g. error messages from scanner and
    parser)
  wxStreamToTextRedirector *outputTextRedirect;
  SwitchesCheckListBox *switchesCtrl;
  wxPanel *simctrls_container;
  wxButton *simctrl_run;
  wxButton *simctrl_continue;
  wxButton *monitors_add_btn;
  wxButton *monitors_rem_btn;
  circuit* c;
```

```
41
42    void OnExit(wxCommandEvent& event);     // callback for exit menu
        item
43    void OnAbout(wxCommandEvent& event);    // callback for about menu
        item
44    void OnOpenFile(wxCommandEvent& event); // callback for open file
        menu item
45    void OnButtonRun(wxCommandEvent& event);
46    void OnButtonContinue(wxCommandEvent& event);
47    void OnButtonAddMon(wxCommandEvent& event);
48    void OnButtonDelMon(wxCommandEvent& event);
49    void OnButtonEditDevs(wxCommandEvent& event);
50    void UpdateControlStates();
51    void OnMenuClearCircuit(wxCommandEvent &event);
52
53    DECLARE_EVENT_TABLE()
54 };
55
56
57 class AddMonitorsDialog: public wxDialog
58 {
59 public:
60    AddMonitorsDialog(circuit* circ, wxWindow* parent, const wxString&
        title, const wxPoint& pos = wxDefaultPosition, const wxSize& size
        = wxDefaultSize, long style = wxDEFAULT_DIALOG_STYLE);
61 private:
62    circuit* c;
63    int oldMonCount;
64    CircuitElementInfoVector availableOutputs;
65    wxListBox *lbox;
66    void OnOK(wxCommandEvent& event);
67    DECLARE_EVENT_TABLE()
68 };
69
70 class DelMonitorsDialog: public wxDialog
71 {
72 public:
73    DelMonitorsDialog(circuit* circ, wxWindow* parent, const wxString&
        title, const wxPoint& pos = wxDefaultPosition, const wxSize& size
        = wxDefaultSize, long style = wxDEFAULT_DIALOG_STYLE);
74 private:
75    circuit* c;
76    wxListBox *lbox;
77    void OnOK(wxCommandEvent& event);
78    DECLARE_EVENT_TABLE()
79 };
80
81 #endif /* gui_h */
```

Listing 13: gui.cc

```
1  #include "gui.h"
2  #include "wx_icon.xpm"
3  #include <iostream>
4  #include <vector>
5  #include <algorithm>
6  #include "scanner.h"
7  #include "parser.h"
8  #include "gui-devices.h"
9  #include "gui-id.h"
10
11 using namespace std;
12
13
14 // MyFrame
       ////////////////////////////////////////////////////////////////////////
15
16
17 BEGIN_EVENT_TABLE(MyFrame, wxFrame)
18   EVT_MENU(wxID_EXIT, MyFrame::OnExit)
19   EVT_MENU(wxID_ABOUT, MyFrame::OnAbout)
20   EVT_MENU(MENU_CLEAR_CIRCUIT, MyFrame::OnMenuClearCircuit)
21   EVT_MENU(wxID_OPEN, MyFrame::OnOpenFile)
22   EVT_BUTTON(SIMCTRL_BUTTON_RUN_ID, MyFrame::OnButtonRun)
23   EVT_BUTTON(SIMCTRL_BUTTON_CONT_ID, MyFrame::OnButtonContinue)
24   EVT_BUTTON(MONITORS_ADD_BUTTON_ID, MyFrame::OnButtonAddMon)
25   EVT_BUTTON(MONITORS_DEL_BUTTON_ID, MyFrame::OnButtonDelMon)
26   EVT_BUTTON(DEVICES_EDIT_BUTTON_ID, MyFrame::OnButtonEditDevs)
27 END_EVENT_TABLE()
28
29 MyFrame::MyFrame(wxWindow *parent, const wxString& title, const
       wxPoint& pos, const wxSize& size,
30                  names *names_mod, devices *devices_mod, monitor *
       monitor_mod, network *net_mod, long style):
31   wxFrame(parent, wxID_ANY, title, pos, size, style)
32   // Constructor - initialises pointers to names, devices and monitor
       classes, lays out widgets
33   // using sizers
34 {
35   SetIcon(wxIcon(wx_icon));
36
37   if (names_mod == NULL || devices_mod == NULL || monitor_mod == NULL
       || net_mod == NULL) {
38     cout << "Cannot operate GUI without names, devices, network and
       monitor classes" << endl;
39     exit(1);
40   }
41   c = new circuit(names_mod, devices_mod, monitor_mod, net_mod);
42
43   // Menu items
44   wxMenu *fileMenu = new wxMenu;
```

```
45  fileMenu->Append(wxID_OPEN);
46  fileMenu->Append(MENU_CLEAR_CIRCUIT, _("Clear circuit"));
47  fileMenu->AppendSeparator();
48  fileMenu->Append(wxID_ABOUT, _("&About"));
49  fileMenu->Append(wxID_EXIT, _("&Quit"));
50  wxMenuBar *menuBar = new wxMenuBar;
51  menuBar->Append(fileMenu, _("&File"));
52  SetMenuBar(menuBar);
53
54  // Everything is contained in a wxPanel for improved appearance in
      wine/windows, as
55  // described in http://wiki.wxwidgets.org/WxFAQ
56  wxPanel* mainPanel = new wxPanel(this, wxID_ANY, wxDefaultPosition,
       wxDefaultSize, wxTAB_TRAVERSAL);
57  wxBoxSizer *topsizer = new wxBoxSizer(wxHORIZONTAL);
58  wxBoxSizer *leftsizer = new wxBoxSizer(wxVERTICAL);
59
60  // Canvas for drawing monitor traces
61  canvas = new MyGLCanvas(c, mainPanel, wxID_ANY, wxDefaultPosition,
      wxDefaultSize, wxBORDER_SUNKEN);
62  leftsizer->Add(canvas, 3, wxEXPAND | wxALL, 10);
63
64  // Create the log textbox, mainly for displaying error messages
      from the parser, captures everything sent to cout
65  // wxTE_DONTWRAP means that a horizontal scrollbar will be used
      instead of wrapping, so that the position of an error can be
      indicated correctly
66  outputTextCtrl = new wxTextCtrl(mainPanel, OUTPUT_TEXTCTRL_ID, wxT(
      ""), wxDefaultPosition, wxDefaultSize, wxTE_MULTILINE |
      wxTE_READONLY | wxTE_DONTWRAP);
67
68  // Set log textbox to a monospace font, so that the position of an
      error can be indicated correctly
69  wxTextAttr outputTextAttr = outputTextCtrl->GetDefaultStyle();
70  outputTextAttr.SetFont(wxFont(10, wxFONTFAMILY_MODERN,
      wxFONTSTYLE_NORMAL, wxFONTWEIGHT_NORMAL));
71  outputTextCtrl->SetDefaultStyle(outputTextAttr);
72
73  // Redirect all text sent to cout to the log textbox
74  outputTextRedirect = new wxStreamToTextRedirector(outputTextCtrl);
75  leftsizer->Add(outputTextCtrl, 1, wxEXPAND | wxALL, 10);
76  topsizer->Add(leftsizer, 4, wxEXPAND | wxALL, 10);
77
78  // Simulation controls box
79  wxBoxSizer *sidesizer = new wxBoxSizer(wxVERTICAL);
80  simctrls_container = new wxPanel(mainPanel, wxID_ANY);
81  wxStaticBoxSizer *simctrls_sizer = new wxStaticBoxSizer(wxVERTICAL,
       simctrls_container, _("Simulation"));
82  wxBoxSizer *simctrls_cycles_sizer = new wxBoxSizer(wxHORIZONTAL);
83  wxBoxSizer *simctrls_button_sizer = new wxBoxSizer(wxHORIZONTAL);
84
```

```
85  // Run and continue simulation buttons
86  simctrl_run = new wxButton(simctrls_container,
      SIMCTRL_BUTTON_RUN_ID, _("Run"));
87  simctrl_continue = new wxButton(simctrls_container,
      SIMCTRL_BUTTON_CONT_ID, _("Continue"));
88  simctrls_button_sizer->Add(simctrl_run, 0, wxALL, 10);
89  simctrls_button_sizer->Add(simctrl_continue, 0, wxALL, 10);
90  // Simulation cycle number spinner
91  simctrls_cycles_sizer->Add(new wxStaticText(simctrls_container,
      wxID_ANY, _("Cycles")), 0, wxALL|wxALIGN_CENTER_VERTICAL, 10);
92  spin = new wxSpinCtrl(simctrls_container, MY_SPINCNTRL_ID, wxString
      (wxT("42")));
93  spin->SetRange(1,1000000);
94  simctrls_cycles_sizer->Add(spin, 0, wxALL, 10);
95
96  simctrls_sizer->Add(simctrls_cycles_sizer);
97  simctrls_sizer->Add(simctrls_button_sizer);
98  simctrls_container->SetSizerAndFit(simctrls_sizer);
99
100 // Buttons to open add/remove monitor dialogs
101 wxStaticBoxSizer *edit_sizer = new wxStaticBoxSizer(wxVERTICAL,
      mainPanel, _("Edit circuit"));
102 monitors_add_btn = new wxButton(mainPanel, MONITORS_ADD_BUTTON_ID,
      _("Add monitors"));
103 monitors_rem_btn = new wxButton(mainPanel, MONITORS_DEL_BUTTON_ID,
      _("Remove monitors"));
104 edit_sizer->Add(monitors_add_btn, 0, (wxALL & ~wxBOTTOM) | wxEXPAND
      , 10);
105 edit_sizer->Add(monitors_rem_btn, 0, wxLEFT | wxRIGHT | wxEXPAND,
      10);
106 edit_sizer->Add(new wxButton(mainPanel, DEVICES_EDIT_BUTTON_ID, _("
      Edit devices")), 0, (wxALL & ~wxTOP) | wxEXPAND, 10);
107
108 // wxCheckListBox that allows switch states to be changed
109 wxStaticBoxSizer *switches_sizer = new wxStaticBoxSizer(wxVERTICAL,
       mainPanel, _("Switches"));
110 switchesCtrl = new SwitchesCheckListBox(c, mainPanel,
      SWITCHES_CTRL_ID, wxDefaultPosition, wxDefaultSize,
      wxLB_NEEDED_SB);
111 switches_sizer->Add(switchesCtrl, 1, wxEXPAND | wxALL, 10);
112
113
114 sidesizer->Add(simctrls_container, 0, wxALL, 10);
115 sidesizer->Add(edit_sizer, 0, wxEXPAND | wxALL, 10);
116 sidesizer->Add(switches_sizer, 1, wxEXPAND | wxALL, 10);
117 topsizer->Add(sidesizer, 0, wxEXPAND | wxALIGN_CENTER);
118
119 SetSizeHints(400, 400);
120 mainPanel->SetSizer(topsizer);
121
122 c->circuitChanged.Attach(this, &MyFrame::UpdateControlStates);
```

```
  c->monitorsChanged.Attach(this, &MyFrame::UpdateControlStates);
  c->monitorSamplesChanged.Attach(this, &MyFrame::UpdateControlStates
    );
  UpdateControlStates();
}

MyFrame::~MyFrame()
{
  c->circuitChanged.Detach(this);
  c->monitorsChanged.Detach(this);
  c->monitorSamplesChanged.Detach(this);
  delete outputTextRedirect;
}

void MyFrame::OnExit(wxCommandEvent &event)
// Callback for the exit menu item
{
  Close(true);
}

void MyFrame::OnAbout(wxCommandEvent &event)
// Callback for the about menu item
{
  wxMessageDialog about(this, _("Logic simulator\nIIA GF2 Team 8\
    n2013"), _("About Logsim"), wxICON_INFORMATION | wxOK);
  about.ShowModal();
}

void MyFrame::OnMenuClearCircuit(wxCommandEvent &event)
{
  c->Clear();
}

void MyFrame::OnOpenFile(wxCommandEvent &event)
// Callback for the File -> Open menu item
{
  wxFileDialog openFileDialog(this, _("Open logic circuit"), wxT("../
    examples/"), wxT(""), _("Logic circuit files (*.gf2)|*.gf2|All
    files (*.*)|*.*"), wxFD_OPEN | wxFD_FILE_MUST_EXIST |
    wxFD_CHANGE_DIR);
  if (openFileDialog.ShowModal() == wxID_CANCEL)
    return; // cancelled, don't open a file
  loadFile(openFileDialog.GetPath().mb_str());
}

bool MyFrame::loadFile(const char * filename)
// load a file (can be called by menu File->Open or for the command
    line argument)
{
  bool result; //True if file is opened correctly
  // Clear log window
  outputTextCtrl->ChangeValue(wxT(""));
  cout << "Loading file " << filename << endl;

  c->Clear();

  scanner *smz = new scanner(c->nmz(), filename, result);
  error *erz = new error(smz);
  parser *pmz = new parser(c->netz(), c->dmz(), c->mmz(), smz, erz);
  if (result) result = pmz->readin();

  if (result)
  {
    c->netz()->checknetwork(result);
  }

  if (!result)
  {
    cout << "Failed to load file" << endl;
    c->Clear();

    // scroll to the start of the output so that the first error
    message (which may have caused any subsequent error messages) is
    visible
    outputTextCtrl->ShowPosition(0);
  }

  c->circuitChanged.Trigger();
  c->monitorsChanged.Trigger();
  c->monitorSamplesChanged.Trigger();

  if (!result)
  {
    canvas->SetErrorMessage(_("Failed to load file"));
  }

  delete pmz;
  delete erz;
  delete smz;

  return result;
}

void MyFrame::UpdateControlStates()
{
  // Update enabled/disabled state of controls
  if (c->netz()->devicelist()==NULL)
  {
    // If there are no devices, disable simulation controls and add/
    remove monitor buttons
    simctrls_container->Disable();
    monitors_add_btn->Disable();
```

```
    monitors_rem_btn->Disable();
  }
  else
  {
    // The circuit contains some devices, so enable the simulation controls if all inputs are connected
    // It is assumed that unconnected inputs are listed by whichever bit of code has allowed them to exist in the circuit, this function just updates control stated
    bool ok;
    c->netz()->checknetwork(ok, true);
    if (ok)
      simctrls_container->Enable();
    else
      simctrls_container->Disable();
    // Only enable the add monitors button if some unmonitored outputs exist
    if (c->GetUnmonitoredOutputs())
      monitors_add_btn->Enable();
    else
      monitors_add_btn->Disable();
    // Only enable the remove monitors button if some monitors exist
    if (c->mmz()->moncount()>0)
      monitors_rem_btn->Enable();
    else
      monitors_rem_btn->Disable();
    // Disable the continue button if the run button has not been used first (so GetTotalCycles returns 0) or there are some new monitors (so some monitors do not contain any samples)
    bool someEmpty = (c->GetTotalCycles()==0);
    for (int i=0; i<c->mmz()->moncount(); i++)
    {
      if (c->mmz()->getsamplecount(i)==0)
        someEmpty = true;
    }
    if (someEmpty)
      simctrl_continue->Disable();
    else
      simctrl_continue->Enable();

  }
}

void MyFrame::OnButtonRun(wxCommandEvent &event)
// Callback for the run simulation button
{
  c->Simulate(spin->GetValue(),true);
  simctrl_continue->Enable();
}

void MyFrame::OnButtonContinue(wxCommandEvent &event)
// Callback for the continue simulation button
{
  c->Simulate(spin->GetValue(),false);
}

void MyFrame::OnButtonAddMon(wxCommandEvent& event)// "Add monitors" button clicked
{
  int oldMonCount = c->mmz()->moncount();
  AddMonitorsDialog *dlg = new AddMonitorsDialog(c, this, _("Add monitors"), wxDefaultPosition, wxDefaultSize, wxDEFAULT_DIALOG_STYLE | wxRESIZE_BORDER);
  if (dlg->ShowModal()==wxID_OK && oldMonCount!=c->mmz()->moncount())
  {
    if (c->GetTotalCycles())
    {
      cout << wxString(wxPLURAL("Monitor added, run simulation again to see updated signals", "Monitors added, run simulation again to see updated signals", c->mmz()->moncount()-oldMonCount)).mb_str() << endl;
    }
  }
  dlg->Destroy();
}

void MyFrame::OnButtonDelMon(wxCommandEvent& event)// "Remove monitors" button clicked
{
  DelMonitorsDialog *dlg = new DelMonitorsDialog(c, this, _("Remove monitors"), wxDefaultPosition, wxDefaultSize, wxDEFAULT_DIALOG_STYLE | wxRESIZE_BORDER);
  dlg->ShowModal();
  dlg->Destroy();
}

void MyFrame::OnButtonEditDevs(wxCommandEvent& event)// "Edit devices" button clicked
{
  DevicesDialog *dlg = new DevicesDialog(c, this, wxID_ANY, _("Edit devices"));
  dlg->ShowModal();
  dlg->Destroy();
  bool ok;
  c->netz()->checknetwork(ok);
  if (!ok) canvas->SetErrorMessage(_("Circuit contains unconnected inputs"));
  else canvas->ClearErrorMessage();
  UpdateControlStates();
}
```

```
300
301 AddMonitorsDialog::AddMonitorsDialog(circuit* circ, wxWindow* parent,
        const wxString& title, const wxPoint& pos, const wxSize& size,
      long style):
302   wxDialog(parent, wxID_ANY, title, pos, size, style)
303 {
304   SetIcon(wxIcon(wx_icon));
305
306   c = circ;
307   oldMonCount = c->mmz()->moncount();
308
309   // Make a list of unmonitored outputs
310   c->GetUnmonitoredOutputs(&availableOutputs);
311   sort(availableOutputs.begin(), availableOutputs.end(),
       CircuitElementInfo_namestrcmp);
312   wxArrayString displayedOutputs;
313   CircuitElementInfoVector_to_wxArrayString(availableOutputs,
       displayedOutputs);
314
315   wxBoxSizer* topsizer = new wxBoxSizer(wxVERTICAL);
316   topsizer->Add(new wxStaticText(this, wxID_ANY, _("Select output(s)
       to monitor:")), 0, wxLEFT | wxRIGHT | wxTOP, 10);
317   lbox = new wxListBox(this, wxID_ANY, wxDefaultPosition,
       wxDefaultSize, displayedOutputs, wxLB_EXTENDED | wxLB_NEEDED_SB);
318   topsizer->Add(lbox, 1, wxALL | wxEXPAND, 10);
319   topsizer->Add(CreateButtonSizer(wxOK | wxCANCEL), 0, wxALL |
       wxEXPAND, 10);
320   SetSizerAndFit(topsizer);
321 }
322
323 void AddMonitorsDialog::OnOK(wxCommandEvent& event)
324 {
325   wxArrayInt selections;
326   lbox->GetSelections(selections);
327   int num = selections.GetCount();
328   bool ok;
329   for (int i=0; i<num; i++)
330   {
331     c->mmz()->makemonitor(availableOutputs[selections[i]].d->id,
       availableOutputs[selections[i]].o->id, ok);
332   }
333   if (c->mmz()->moncount() != oldMonCount)
334   {
335     c->monitorsChanged.Trigger();
336   }
337   EndModal(wxID_OK);
338 }
339
340 BEGIN_EVENT_TABLE(AddMonitorsDialog, wxDialog)
341   EVT_BUTTON(wxID_OK, AddMonitorsDialog::OnOK)
342 END_EVENT_TABLE()
```

```
343
344
345 DelMonitorsDialog::DelMonitorsDialog(circuit* circ, wxWindow* parent,
        const wxString& title, const wxPoint& pos, const wxSize& size,
      long style):
346   wxDialog(parent, wxID_ANY, title, pos, size, style)
347 {
348   SetIcon(wxIcon(wx_icon));
349   c = circ;
350
351   // Copy monitor names into a wxArrayString to pass to the listbox
352   int monCount = c->mmz()->moncount();
353   wxArrayString monitorList;
354   monitorList.Alloc(monCount);
355   for (int i=0; i<monCount; i++)
356   {
357     monitorList.Add(wxString(c->mmz()->getsignalstring(i).c_str(),
       wxConvUTF8));
358   }
359
360   // Create controls
361   wxBoxSizer* topsizer = new wxBoxSizer(wxVERTICAL);
362   topsizer->Add(new wxStaticText(this, wxID_ANY, _("Select monitors
       to remove:")), 0, wxLEFT | wxRIGHT | wxTOP, 10);
363   lbox = new wxListBox(this, wxID_ANY, wxDefaultPosition,
       wxDefaultSize, monitorList, wxLB_EXTENDED | wxLB_NEEDED_SB);
364   topsizer->Add(lbox, 1, wxALL | wxEXPAND, 10);
365   topsizer->Add(CreateButtonSizer(wxOK | wxCANCEL), 0, wxALL |
       wxEXPAND, 10);
366   SetSizerAndFit(topsizer);
367 }
368
369 void DelMonitorsDialog::OnOK(wxCommandEvent& event)
370 {
371   wxArrayInt selections;
372   lbox->GetSelections(selections);
373   bool ok;
374   monitor* mmz = c->mmz();
375   name dev, outp;
376   // Remove selected monitors
377   // Note loop direction - remmonitor deletes an entry and shifts
       monitors into the space, changing the IDs of later monitors.
       Therefore remove monitors with a higher ID (towards the end of
       the listbox) first.
378   for (int i=selections.GetCount()-1; i>=0; i--)
379   {
380     mmz->getmonname(selections[i], dev, outp);
381     mmz->remmonitor(dev, outp, ok);
382   }
383   if (selections.GetCount())
384     c->monitorsChanged.Trigger();
```

```
385
386   EndModal(wxID_OK);
387 }
388
389 BEGIN_EVENT_TABLE(DelMonitorsDialog, wxDialog)
390   EVT_BUTTON(wxID_OK, DelMonitorsDialog::OnOK)
391 END_EVENT_TABLE()
```

3.2.4 Main GUI: monitor traces canvas

Listing 14: gui-canvas.h

```
1  #ifndef gui_canvas_h
2  #define gui_canvas_h
3
4  #include "circuit.h"
5  #include "gui-misc.h"
6  #include <wx/wx.h>
7  #include <wx/glcanvas.h>
8
9  int GetGlutTextWidth(wxString txt, void *font=NULL);
10 void DrawGlutText(int x, int y, wxString txt, void *font=NULL);
11
12 class MyGLCanvas;
13
14 void wxRect_GlVertex(const wxRect& r);
15
16 class GLCanvasMonitorTrace
17 {
18 public:
19   GLCanvasMonitorTrace();
20   GLCanvasMonitorTrace(int newMonId, circuit* c);
21   // Get or set the monitor id (the "n" in "n'th monitor" in monitor
       class calls, also determines position)
22   int GetMonitorId();
23   void SetMonitorId(int newMonId);
24   // Notify of a change in the number of displayed cycles
25   void OnMonitorSamplesChanged(int totalCycles_new, int
       continuedCycles_new);
26   // Draw the signal trace, visible coordinates are used so that time
        is not wasted in drawing areas hidden due to scrolling.
27   void Draw(MyGLCanvas *canvas, const wxRect& visibleRegion);
28   void DrawName(MyGLCanvas *canvas, const wxRect& visibleRegion);
29   // Get the width in pixels of the name, used by MyGLCanvas.Render()
        to determine how much space to leave between the traces and the
       edge of the canvas
30   int GetNameWidth();
31   // Set the geometry and positioning of the monitor trace.
32   // xOffset - the x position of the graph origin
33   // yCentre - the y position of the centre of the drawn signal (
       halfway between high and low signal levels)
34   // xScale - the x-axis scale (the number of pixels per cycle).
35   // height - the height of the signal trace line itself,
36   // padding - the distance between the top of the signal trace and
       the edge of the graph background
37   // spacing - the vertical distance between centre lines of
       consecutive monitor traces.
38   // xBgName - the distance between the edge of the screen and the
       faint signal name drawn on top of the traces when the text to the
        left is invisible.
39   void SetGeometry(int xOffset_new, int yCentre_new, double
       xScale_new, int sigHeight_new, int padding_new, int spacing_new,
       int xBgName_new, int axisLabelInterval_new);
40 private:
41   int monId;
42   circuit* c;
43   wxString monName;// cached monitor name, to avoid constructing it
       again every time the signal is drawn
44   int monNameWidth;// width in pixels of monitor name
45   bool geometrySet;// whether SetGeometry() has been called
46
47   // variables controlling position and size of the trace, see
       SetGeometry() for details
48   int xOffset, yCentre, sigHeight, padding, spacing, xBgName;
49   int axisLabelInterval;// cycles between numbers on x axis
50   double xScale;
51
52   void UpdateName();// Update monName and monNameWidth
53   int continuedCycles;// how many simulation cycles were completed
       last time the run or continue button was used
54   int totalCycles;// how many simulation cycles have been completed
       in total
55 };
56
57
58 class MyGLCanvas: public wxGLCanvas, public wxScrollHelperNative
59 {
60 public:
61   MyGLCanvas(circuit* circ, wxWindow *parent, wxWindowID id =
       wxID_ANY, const wxPoint& pos = wxDefaultPosition, const wxSize&
       size = wxDefaultSize, long style = 0, const wxString& name = wxT(
       "MyGLCanvas"));
62   ~MyGLCanvas();
63   void Render(); // function to draw canvas contents
64   void OnMonitorSamplesChanged();
65   void OnMonitorsChanged();
66   void UpdateMinCanvasSize();
67   void SetErrorMessage(wxString txt);
68   void ClearErrorMessage();
```

```
private:
  bool init;                          // has the GL context been
    initialised?
  circuit* c;
  int maxMonNameWidth;
  vector<GLCanvasMonitorTrace> mons;
  void InitGL();                      // function to initialise GL
    context
  void OnSize(wxSizeEvent& event);    // callback for when canvas is
    resized
  void OnPaint(wxPaintEvent& event); // callback for when canvas is
    exposed
  void OnMouse(wxMouseEvent& event); // callback for mouse events
    inside canvas
  void DrawInfoTextCentre(wxString txt, bool isError = false);
  void Redraw();
  wxString errorMessage;
  int scrollX, scrollY;
  int minXScale, maxXScale;
public:
  virtual void ScrollWindow(int dx, int dy, const wxRect* rect = (
    wxRect *)NULL);

  WX_FORWARD_TO_SCROLL_HELPER() // copied from wxScrolledWindow

private:
  DECLARE_EVENT_TABLE()
};



#endif /* gui_canvas_h */
```

Listing 15: gui-canvas.cc

```
#include "gui-canvas.h"
#include <GL/glut.h>
#include <cmath>

// Get the width in pixels of some text when drawn using a particular
    font (font defaults to GLUT_BITMAP_HELVETICA_12)
int GetGlutTextWidth(wxString txt, void *font)
{
  int width = 0;
  if (!font) font = GLUT_BITMAP_HELVETICA_12;
  for (int i = 0; i < txt.Len(); i++)
    width += glutBitmapWidth(font, txt[i]);
  return width;
}

// Draws some text at the given position (font defaults to
    GLUT_BITMAP_HELVETICA_12)
void DrawGlutText(int x, int y, wxString txt, void *font)
{
  if (!font) font = GLUT_BITMAP_HELVETICA_12;
  int xMove = 0, yMove = 0;
  /* If x or y are outside the viewport, call glRasterPos2f with a
    position inside
   * the viewport, then move it using glBitmap (as described in
    glRasterPos man page).
   * Otherwise the text doesn't get rendered.
   * Assuming viewport starts at (0,0) and only implemented for
    positions to the left of
   * or above (0,0) for now. */
  if (x<0) xMove = x;
  if (y<0) yMove = y;
  glRasterPos2f(x-xMove, y-yMove);
  if (xMove || yMove) glBitmap (0, 0, 0, 0, xMove, yMove, NULL);
  for (int i = 0; i < txt.Len(); i++)
    glutBitmapCharacter(font, txt[i]);
}

// Call glVertex for the positions of the corners of a wxRect
// This makes drawing boxes a bit simpler
void wxRect_GlVertex(const wxRect& r)
{
  glVertex2i(r.x, r.y);
  glVertex2i(r.x+r.width, r.y);
  glVertex2i(r.x+r.width, r.y+r.height);
  glVertex2i(r.x, r.y+r.height);
}

// GLCanvasMonitorTrace - class to handle drawing of one monitor
    trace
GLCanvasMonitorTrace::GLCanvasMonitorTrace() :
  monId(-1), c(NULL), monName(wxT("")), monNameWidth(0), geometrySet(
    false)
{}

GLCanvasMonitorTrace::GLCanvasMonitorTrace(int newMonId, circuit*
    circ) :
  geometrySet(false)
{
  c = circ;
  SetMonitorId(newMonId);
}

int GLCanvasMonitorTrace::GetMonitorId()
{
  return monId;
}
```

```
59
60  void GLCanvasMonitorTrace::SetMonitorId(int newMonId)
61  {
62    monId = newMonId;
63    UpdateName();
64  }
65
66  void GLCanvasMonitorTrace::OnMonitorSamplesChanged(int
        totalCycles_new, int continuedCycles_new)
67  {
68    totalCycles = totalCycles_new;
69    continuedCycles = continuedCycles_new;
70  }
71
72  void GLCanvasMonitorTrace::UpdateName()
73  {
74    if (!c)
75    {
76      monName = wxT("");
77      monNameWidth = 0;
78      return;
79    }
80    monName = wxString(c->mmz()->getsignalstring(monId).c_str(),
        wxConvUTF8);
81    monNameWidth = GetGlutTextWidth(monName, GLUT_BITMAP_HELVETICA_12);
82  }
83
84  int GLCanvasMonitorTrace::GetNameWidth()
85  {
86    return monNameWidth;
87  }
88
89  void GLCanvasMonitorTrace::SetGeometry(int xOffset_new, int
        yCentre_new, double xScale_new, int sigHeight_new, int
        padding_new, int spacing_new, int xBgName_new, int
        axisLabelInterval_new)
90  {
91    geometrySet = true;
92    xOffset = xOffset_new;
93    yCentre = yCentre_new;
94    xScale = xScale_new;
95    sigHeight = int(sigHeight_new/2)*2;// make sure this is even, so
        that the trace is symmetrical about the horizontal centre line
96    padding = padding_new;
97    spacing = spacing_new;
98    xBgName = xBgName_new;
99    axisLabelInterval = axisLabelInterval_new;
100 }
101
102 void GLCanvasMonitorTrace::Draw(MyGLCanvas *canvas, const wxRect&
        visibleRegion)
103 {
104   if (!c || !canvas || monId<0 || monId>=c->mmz()->moncount() || !
        geometrySet) return;
105
106   // If this monitor was added after the simulation was run, it might
        have no data even though c->GetTotalCycles() is greater than
        zero
107   int sampleCount = c->mmz()->getsamplecount(monId);
108   if (!sampleCount)
109   {
110     glColor4f(0.8, 0.0, 0.0, 1.0);
111     if (xOffset < visibleRegion.x+xBgName)
112     {
113       DrawGlutText(xBgName, yCentre-5, _("No data"),
        GLUT_BITMAP_HELVETICA_12);
114     }
115     else
116     {
117       DrawGlutText(xOffset+10, yCentre-5, _("No data"),
        GLUT_BITMAP_HELVETICA_12);
118     }
119   }
120   if (totalCycles > sampleCount)
121     totalCycles = sampleCount;
122   if (continuedCycles > sampleCount)
123     continuedCycles = sampleCount;
124
125   wxRect backgroundRegion(xOffset, yCentre-sigHeight/2-padding, ceil(
        xScale*totalCycles), sigHeight+padding*2);
126   wxRect traceRegion = wxRect(xOffset, yCentre-sigHeight/2-padding
        -11, ceil(xScale*totalCycles), sigHeight+padding*2+11);//
        includes cycle numbers on the axis
127   if (traceRegion.Intersect(visibleRegion).IsEmpty()) return;
128   wxRect clippedbg = backgroundRegion;
129   clippedbg.Intersect(visibleRegion);
130
131   // background colour
132   glBegin(GL_QUADS);
133   glColor4f(0.0, 0.5, 0.0, 0.08);
134   wxRect_GlVertex(clippedbg);
135   glEnd();
136
137   // border
138   glColor4f(0.0, 0.7, 0.0, 0.4);
139   glBegin(GL_LINE_LOOP);
140   wxRect_GlVertex(backgroundRegion);
141   glEnd();
142
143   // actual signal trace
144   if (xScale>5) glLineWidth(2);
145   glBegin(GL_LINE_STRIP);
```

```
  glColor4f(0.0, 0.8, 0.0, 1.0);
  int y1, y2, i;
  asignal s;
  int firstCycle = 0;
  int cycleLimit = totalCycles;
  if (xOffset < visibleRegion.x)
    firstCycle = int((visibleRegion.x-xOffset)/xScale);
  if (xOffset + totalCycles*xScale > visibleRegion.x+visibleRegion.
    width)
    cycleLimit = int((visibleRegion.x+visibleRegion.width - xOffset)/
    xScale) + 1;
  if (cycleLimit>totalCycles)
    cycleLimit = totalCycles;
  int prevY = yCentre;
  for (i=firstCycle; i<cycleLimit; i++)
  {
    if (c->mmz()->getsignaltrace(monId, i, s))
    {
      if (s==low || s==rising)
        y1 = yCentre-sigHeight/2;
      else
        y1 = yCentre+sigHeight/2;
      if (s==low || s==falling)
        y2 = yCentre-sigHeight/2;
      else
        y2 = yCentre+sigHeight/2;
      if (y1!=prevY) glVertex2i(xOffset+xScale*i, y1);
      glVertex2i(xOffset+xScale*(i+1), y2);
      prevY = y2;
    }
  }
  glEnd();
  glLineWidth(1);

  // Draw cycle numbers on axis and dashed vertical lines for them
  firstCycle = (int(firstCycle/axisLabelInterval)-1) *
    axisLabelInterval;
  if (firstCycle<0)
    firstCycle = 0;
  cycleLimit = (int(cycleLimit/axisLabelInterval)+1) *
    axisLabelInterval;
  if (cycleLimit>totalCycles)
    cycleLimit = totalCycles;
  glLineStipple(2, 0xAAAA);
  glEnable(GL_LINE_STIPPLE);
  for (i=firstCycle; i<=cycleLimit; i+=axisLabelInterval)
  {
    if (i!=0 && i!=totalCycles)
    {
      glBegin(GL_LINE_STRIP);
      glColor4f(0.0, 0.0, 0.0, 0.2);
      glVertex2i(xOffset+xScale*i, yCentre-sigHeight/2-padding);
      glVertex2i(xOffset+xScale*i, yCentre+sigHeight/2+padding);
      glEnd();
    }
    glColor4f(0.0, 0.0, 0.0, 1.0);
    wxString labelText;
    labelText.Printf(wxT("%d"), i);
    int labelWidth = GetGlutTextWidth(labelText,
    GLUT_BITMAP_HELVETICA_10);
    DrawGlutText(xOffset+xScale*i - labelWidth/2, yCentre-sigHeight
    /2-padding-11, labelText, GLUT_BITMAP_HELVETICA_10);
  }
  glDisable(GL_LINE_STIPPLE);
}

void GLCanvasMonitorTrace::DrawName(MyGLCanvas *canvas, const wxRect&
     visibleRegion)
{
  if (!geometrySet) return;
  if (xOffset < visibleRegion.x+xBgName && c->mmz()->getsamplecount(
    monId))
  {
    glColor4f(0.0, 0.0, 0.0, 0.4);
    DrawGlutText(xBgName, yCentre-5, monName,
    GLUT_BITMAP_HELVETICA_12);
  }
  else
  {
    glColor4f(0.0, 0.0, 0.0, 1.0);
    DrawGlutText(xOffset-monNameWidth-5, yCentre-5, monName,
    GLUT_BITMAP_HELVETICA_12);
  }
}

// MyGLCanvas
    ////////////////////////////////////////////////////////////////////////


BEGIN_EVENT_TABLE(MyGLCanvas, wxGLCanvas)
  EVT_SIZE(MyGLCanvas::OnSize)
  EVT_PAINT(MyGLCanvas::OnPaint)
  EVT_MOUSE_EVENTS(MyGLCanvas::OnMouse)
END_EVENT_TABLE()

int wxglcanvas_attrib_list[5] = {WX_GL_RGBA, WX_GL_DOUBLEBUFFER,
    WX_GL_DEPTH_SIZE, 16, 0};

MyGLCanvas::MyGLCanvas(circuit* circ, wxWindow *parent, wxWindowID id
    ,
                       const wxPoint& pos, const wxSize& size, long
    style, const wxString& name):
```

```
  wxGLCanvas(parent, id, pos, size, style, name,
    wxglcanvas_attrib_list),
  wxScrollHelperNative(this), scrollX(0), scrollY(0)
  // Constructor - initialises private variables
{
  init = false;
  c = circ;
  if (c)
  {
    c->monitorsChanged.Attach(this, &MyGLCanvas::OnMonitorsChanged);
    c->monitorSamplesChanged.Attach(this, &MyGLCanvas::
    OnMonitorSamplesChanged);
    c->monitorSamplesChanged.Attach(this, &MyGLCanvas::
    ClearErrorMessage);
    OnMonitorsChanged();
  }
  SetScrollRate(10,10);
  minXScale = 2;
  maxXScale = 50;
}

MyGLCanvas::~MyGLCanvas()
{
  if (c)
  {
    c->monitorsChanged.Detach(this);
    c->monitorSamplesChanged.Detach(this);
  }
}

void MyGLCanvas::Redraw()
{
  // Redraw the contents of the canvas
  // This might look bizarre, but it seems to work everywhere tested
    (linux, and a cross compiled executable running in wine)
  // Removing any of these calls mean the canvas is sometimes not
    redrawn when running in wine
  Refresh();
  Update();
  Refresh();
}

void MyGLCanvas::ScrollWindow(int dx, int dy, const wxRect *rect)
{
  // override ScrollWindow(), since MyGLCanvas does its own scrolling
     by offsetting all drawn points and clipping
  scrollX += dx;
  scrollY += dy;
  Redraw();
}

```

```
void MyGLCanvas::UpdateMinCanvasSize()
{
  int totalCycles = c->GetTotalCycles();
  if (!mons.size())
    totalCycles = 0;//If all monitors have been deleted, don't
    reserve horizontal space
  // Make sure all the traces fit in the canvas
  int xOffset = maxMonNameWidth+5;
  int maxXTextWidth = ceil(log10(totalCycles))*8;// estimate of max x
     axis scale text width
  SetVirtualSize(minXScale*totalCycles+15+xOffset+maxXTextWidth/2,50*
    mons.size()+10);
}

// Notify of a change to the number of displayed cycles
void MyGLCanvas::OnMonitorSamplesChanged()
{
  int totalCycles = c->GetTotalCycles();
  int continuedCycles = c->GetContinuedCycles();
  for (int i=0; i<mons.size(); i++)
  {
    mons[i].OnMonitorSamplesChanged(totalCycles, continuedCycles);
  }
  UpdateMinCanvasSize();
  // Scroll to the most recently simulated cycles
  Scroll(GetVirtualSize().GetWidth()-GetClientSize().GetWidth(),-1);
  Redraw();
}

// Notify of a change to the active monitors
void MyGLCanvas::OnMonitorsChanged()
{
  int monCount = c->mmz()->moncount();
  mons.resize(monCount);
  maxMonNameWidth = 0;
  int totalCycles = c->GetTotalCycles();
  int continuedCycles = c->GetContinuedCycles();
  for (int i=0; i<monCount; i++)
  {
    mons[i] = GLCanvasMonitorTrace(i, c);
    mons[i].OnMonitorSamplesChanged(totalCycles, continuedCycles);
    if (mons[i].GetNameWidth()>maxMonNameWidth)
      maxMonNameWidth = mons[i].GetNameWidth();
  }
  UpdateMinCanvasSize();
  Redraw();
}

void MyGLCanvas::Render()
{
  unsigned int i;
```

```
326
327   SetCurrent();
328   if (!init)
329   {
330     InitGL();
331     init = true;
332   }
333   glClear(GL_COLOR_BUFFER_BIT);
334   if (c->GetTotalCycles() > 0 && c->mmz()->moncount() > 0)
335   {
336     int totalCycles = c->GetTotalCycles();
337     // Scale traces (within limits) to fit the size of the canvas
338     int monCount = c->mmz()->moncount();
339     int canvasHeight = GetClientSize().GetHeight();
340     int canvasWidth = GetClientSize().GetWidth();
341     int spacing = (canvasHeight-10)/monCount;
342     if (spacing>200) spacing = 200;
343     if (spacing<50) spacing = 50;
344     int height = 0.8*(spacing-14);
345     int xOffset = maxMonNameWidth+10;
346     double xScale = double(canvasWidth-xOffset-10)/c->GetTotalCycles
      ();
347     if (xScale<minXScale) xScale = minXScale;
348     if (xScale>50) xScale = 50;
349
350     wxRect visibleRegion(0,0,canvasWidth,canvasHeight);
351
352     // Work out the best interval to use between displayed cycle
      numbers on the x axis
353     // Intervals will be 1, 2, or 5 * pow(10,n) cycles between
      displayed numbers, where n>=0
354     int minAxisLabelIntervalPixels = ceil(ceil(log10(totalCycles))
      *8*2);
355     int base = 1;
356     int baseMultiples[] = {1,2,5};
357     int axisLabelInterval = 1;
358     while (axisLabelInterval<=totalCycles)
359     {
360       for (i=0; i<3; i++)
361       {
362         axisLabelInterval = baseMultiples[i]*base;
363         if (axisLabelInterval*xScale >= minAxisLabelIntervalPixels)
364           break;
365       }
366       if (axisLabelInterval*xScale >= minAxisLabelIntervalPixels)
367         break;
368       base *= 10;
369     }
370     // Draw the traces
371     for (i=0; i<mons.size(); i++)
372     {
373       mons[i].SetGeometry(xOffset+scrollX, canvasHeight-scrollY-
        spacing*i-spacing/2, xScale, height, spacing*0.075, spacing, 10,
        axisLabelInterval);
374       mons[i].Draw(this, visibleRegion);
375       mons[i].DrawName(this, visibleRegion);
376     }
377     if (errorMessage != wxT(""))
378     {
379       DrawInfoTextCentre(errorMessage, true);
380     }
381   }
382   else if (errorMessage != wxT(""))
383   {
384     DrawInfoTextCentre(errorMessage, true);
385   }
386   else if (c->netz()->devicelist()==NULL)
387   {
388     DrawInfoTextCentre(_("No circuit loaded"));
389   }
390   else if (c->mmz()->moncount()==0)
391   {
392     DrawInfoTextCentre(_("No monitors"));
393   }
394   else
395   {
396     DrawInfoTextCentre(_("No simulation results. Use the run button."
      ));
397   }
398
399   // We've been drawing to the back buffer, flush the graphics
      pipeline and swap the back buffer to the front
400   glFlush();
401   SwapBuffers();
402 }
403
404 void MyGLCanvas::SetErrorMessage(wxString txt)
405 {
406   // Set an error message to be displayed in the centre of the canvas
407   errorMessage = txt;
408   Redraw();
409 }
410
411 void MyGLCanvas::ClearErrorMessage()
412 {
413   errorMessage = wxT("");
414 }
415
416 void MyGLCanvas::DrawInfoTextCentre(wxString txt, bool isError)
417 {
418   // Draw a message in a box in the centre of the canvas
419   // isError makes it a red box
```

```
  int canvasHeight = GetClientSize().GetHeight();
  int canvasWidth = GetClientSize().GetWidth();
  int textWidth = GetGlutTextWidth(txt);
  wxRect background(canvasWidth/2-textWidth/2-15, canvasHeight/2-15,
    textWidth+30, 30);
  if (isError) glColor4f(1.0, 0.85, 0.85, 0.95);
  else glColor4f(0.85, 0.85, 1.0, 0.95);
  glBegin(GL_QUADS);
  wxRect_GlVertex(background);
  glEnd();
  if (isError) glColor4f(0.7, 0.0, 0.0, 0.95);
  else glColor4f(0.0, 0.0, 0.7, 0.95);
  glBegin(GL_LINE_LOOP);
  wxRect_GlVertex(background);
  glEnd();
  glColor4f(0.0, 0.0, 0.0, 1.0);
  DrawGlutText(canvasWidth/2-textWidth/2, canvasHeight/2-4, txt);
}

void MyGLCanvas::InitGL()
// Function to initialise the GL context
{
  int w, h;

  GetClientSize(&w, &h);
  SetCurrent();
  glDrawBuffer(GL_BACK);
  glClearColor(1.0, 1.0, 1.0, 0.0);
  glViewport(0, 0, (GLint) w, (GLint) h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0, w, 0, h, -1, 1);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  glEnable(GL_BLEND);
}

void MyGLCanvas::OnPaint(wxPaintEvent& event)
// Callback function for when the canvas is exposed
{
  wxPaintDC dc(this); // required for correct refreshing under MS
    windows
  Render();
}

void MyGLCanvas::OnSize(wxSizeEvent& event)
// Callback function for when the canvas is resized
{
  wxGLCanvas::OnSize(event); // required on some platforms
  init = false;
  Redraw();// required by some buggy nvidia graphics drivers,
    harmless on other platforms!
}

void MyGLCanvas::OnMouse(wxMouseEvent& event)
// Callback function for mouse events inside the GL canvas
{
  wxString text;
  int w, h;;

  GetClientSize(&w, &h);
  if (event.ButtonDown()) text.Printf(wxT("Mouse button %d pressed at
     %d %d"), event.GetButton(), event.m_x, h-event.m_y);
  if (event.ButtonUp()) text.Printf(wxT("Mouse button %d released at
    %d %d"), event.GetButton(), event.m_x, h-event.m_y);
  if (event.Dragging()) text.Printf(wxT("Mouse dragged to %d %d"),
    event.m_x, h-event.m_y);
  if (event.Leaving()) text.Printf(wxT("Mouse left window at %d %d"),
     event.m_x, h-event.m_y);

  //if (event.ButtonDown() || event.ButtonUp() || event.Dragging() ||
     event.Leaving()) Render(text);
}
```

3.2.5 GUI: miscellaneous widgets and functions

Listing 16: gui-misc.h

```
#ifndef gui_misc_h
#define gui_misc_h

#include <wx/combobox.h>
#include <wx/textctrl.h>
#include <wx/listbox.h>
#include <wx/checklst.h>
#include "circuit.h"

// This file contains misc widgets
// Also some functions for things done in lots of places in the GUI,
    like CircuitElementInfoVector_to_wxArrayString

// CheckListBox to control switch states
class SwitchesCheckListBox: public wxCheckListBox
{
public:
  SwitchesCheckListBox(circuit* circ, wxWindow* parent, wxWindowID id
     = wxID_ANY, const wxPoint& pos = wxDefaultPosition, const wxSize
    & size = wxDefaultSize, long style = 0);
```

```
  ~SwitchesCheckListBox();
  void OnCircuitChanged();
private:
  circuit* c;
  CircuitElementInfoVector switches;
  void OnSwitchChanged(wxCommandEvent& event);
  DECLARE_EVENT_TABLE()
};

extern wxString devicekindstrings[baddevice];

// Dropdown box to select a devicekind, optionally filtering the
    displayed types to those given in filterDevicekinds
class DevicekindDropdown: public wxComboBox
{
public:
  DevicekindDropdown(wxWindow* parent, wxWindowID id = wxID_ANY,
    vector<devicekind> filterDevicekinds = vector<devicekind>());
  devicekind GetDevicekind();
  void SetDevicekind(devicekind dk);
private:
  vector<devicekind> filter;
};

// Text control for entering a device name, with a function for
    validating the new name
class DeviceNameTextCtrl: public wxTextCtrl
{
public:
  DeviceNameTextCtrl(wxWindow* parent, wxWindowID id = wxID_ANY,
    const wxString& value = wxT(""));
  // Validate the new name, optionally showing an error message
  // allowExisting is a name that can be chosen even if a device
    already exists with the same name (so this control can be used
    for editing names of existing devices), use blankname to disallow
     any existing device name
  bool CheckValid(circuit* c,  name allowExisting=blankname, bool
    errorDialog=true);
};

// List of devices (single selection, selection changes are handled
    by the supplied SelectedDevice object)
class DevicesListBox: public wxListBox
{
public:
  DevicesListBox(circuit* circ, SelectedDevice* selectedDev_in,
    wxWindow* parent, wxWindowID id);
  ~DevicesListBox();
  void OnCircuitChanged();
  void OnDeviceSelectionChanged();
  void ReleasePointers();
  devlink GetSelectionAfterDelete();
private:
  SelectedDevice* selectedDev;
  circuit* c;
  CircuitElementInfoVector devs;
  bool isSetting;
  void OnLBoxSelectionChanged(wxCommandEvent& event);
  DECLARE_EVENT_TABLE()
};

// Show an error messagebox
void ShowErrorMsgDialog(wxWindow* parent, wxString txt);

// Put namestrs from a CircuitElementInfoVector into a wxArrayString
void CircuitElementInfoVector_to_wxArrayString(const
    CircuitElementInfoVector& src, wxArrayString& dest);

#endif /* gui_misc_h */
```

Listing 17: gui-misc.cc

```
#include "gui-misc.h"
#include <wx/msgdlg.h>
#include <algorithm>
using namespace std;

SwitchesCheckListBox::SwitchesCheckListBox(circuit* circ, wxWindow*
    parent, wxWindowID id, const wxPoint& pos, const wxSize& size,
    long style)
  : wxCheckListBox(parent, id, pos, size, 0, NULL, style & ~wxLB_SORT
    )
{
  c = circ;
  if (c) c->circuitChanged.Attach(this, &SwitchesCheckListBox::
    OnCircuitChanged);
  OnCircuitChanged();
}

SwitchesCheckListBox::~SwitchesCheckListBox()
{
  if (c) c->circuitChanged.Detach(this);
}

void SwitchesCheckListBox::OnCircuitChanged()
{
  if (!c) return;

  // Update switch names
  switches.resize(0);
  devlink d = c->netz()->devicelist();
  while (d!=NULL)
```

```
  {
    if (d->kind == aswitch) switches.push_back(CircuitElementInfo(d))
    ;
    d = d->next;
  }
  switches.UpdateSignalNames(c);
  wxArrayString switchNames;
  CircuitElementInfoVector_to_wxArrayString(switches, switchNames);
  Set(switchNames);
  // Update switch states
  for (int i=0; i<switches.size(); i++)
  {
    Check(i, switches[i].d->swstate==high);
  }
}

void SwitchesCheckListBox::OnSwitchChanged(wxCommandEvent& event)
{
  if (!c) return;

  // Checkbox changed, update underlying switch state
  int i = event.GetInt();
  if (i>=switches.size())
  {
    ShowErrorMsgDialog(this,  _("Tried to change unknown switch"));
    return;
  }
  if (IsChecked(i))
    switches[i].d->swstate = high;
  else
    switches[i].d->swstate = low;

  c->circuitChanged.Trigger();
}

BEGIN_EVENT_TABLE(SwitchesCheckListBox, wxCheckListBox)
  EVT_CHECKLISTBOX(wxID_ANY, SwitchesCheckListBox::OnSwitchChanged)
END_EVENT_TABLE()


wxString devicekindstrings[baddevice] = {_("Switch"), _("Clock"), _("
    AND gate"), _("NAND gate"), _("OR gate"), _("NOR gate"), _("XOR
    gate"), _("D-type flip-flop")};

DevicekindDropdown::DevicekindDropdown(wxWindow* parent, wxWindowID
    id, vector<devicekind> filterDevicekinds) :
  wxComboBox(parent, id, wxT(""), wxDefaultPosition, wxDefaultSize,
    0, NULL, wxCB_READONLY)
{
  filter = filterDevicekinds;
  // Fill dropdown with available device kinds (optionally filtering
    to only include those listed in filterDevicekinds)
  for (int i=0; i<baddevice; i++)
  {
    if (filter.size()==0 || find(filter.begin(), filter.end(),
    devicekind(i)) != filter.end())
      Append(devicekindstrings[i]);
  }
}

devicekind DevicekindDropdown::GetDevicekind()
{
  for (int i=0; i<baddevice; i++)
  {
    if (devicekindstrings[i] == GetValue())
      return devicekind(i);
  }
  return baddevice;
}

void DevicekindDropdown::SetDevicekind(devicekind dk)
{
  if (dk>=0 && dk<baddevice)
  {
    if (filter.size()==0 || find(filter.begin(), filter.end(), dk) !=
     filter.end())
      SetValue(devicekindstrings[dk]);
  }
}


DeviceNameTextCtrl::DeviceNameTextCtrl(wxWindow* parent, wxWindowID
    id, const wxString& value) :
  wxTextCtrl(parent, id, value)
{}

bool DeviceNameTextCtrl::CheckValid(circuit* c, name allowExisting,
    bool errorDialog)
{
  if (!GetParent()) errorDialog = false;

  if (GetValue() == wxT(""))
  {
    if (errorDialog) ShowErrorMsgDialog(GetParent(), _("Device name
    cannot be blank"));
    return false;
  }
  if (!isalpha(GetValue()[0]))
  {
    if (errorDialog) ShowErrorMsgDialog(GetParent(), _("Device name
    must start with a letter"));
```

```
    return false;
  }
  if (!circuit::IsDeviceNameValid(string(GetValue().mb_str())))
  {
    if (errorDialog) ShowErrorMsgDialog(GetParent(), _("Device name
    must contain only letters and numbers"));
    return false;
  }
  if (c)
  {
    name newname = c->nmz()->cvtname(string(GetValue().mb_str()));
    if (newname!=blankname && newname<=lastreservedname)
    {
      if (errorDialog) ShowErrorMsgDialog(GetParent(), _("Device name
     cannot be a reserved word"));
      return false;
    }
    if (newname!=allowExisting && c->netz()->finddevice(newname)!=
    NULL)
    {
      if (errorDialog) ShowErrorMsgDialog(GetParent(), _("A device
    already exists with that name"));
      return false;
    }
  }
  return true;
}


DevicesListBox::DevicesListBox(circuit* circ, SelectedDevice*
    selectedDev_in, wxWindow* parent, wxWindowID id) :
  wxListBox(parent, id, wxDefaultPosition, wxDefaultSize, 0, NULL,
    wxLB_NEEDED_SB | wxLB_SINGLE)
{
  selectedDev = selectedDev_in;
  c = circ;
  isSetting = false;
  c->circuitChanged.Attach(this, &DevicesListBox::OnCircuitChanged);
  selectedDev->changed.Attach(this, &DevicesListBox::
    OnDeviceSelectionChanged);
  OnCircuitChanged();
}

DevicesListBox::~DevicesListBox()
{
  ReleasePointers();
}

void DevicesListBox::ReleasePointers()
{
  if (selectedDev)
    selectedDev->changed.Detach(this);
  selectedDev = NULL;
  if (c)
    c->circuitChanged.Detach(this);
  c = NULL;
}

void DevicesListBox::OnCircuitChanged()
{
  if (!c || !selectedDev) return;

  devs.resize(0);
  devs.push_back_all_devs(c->netz()->devicelist());
  devs.UpdateSignalNames(c);
  sort(devs.begin(), devs.end(), CircuitElementInfo_namestrcmp);

  int selectedIndex = -1;
  wxArrayString devNames;
  devNames.Alloc(devs.size());
  for (int i=0; i<devs.size(); i++)
  {
    if (devs[i].d==selectedDev->Get())
      selectedIndex = i;
    wxString desc(devs[i].namestr.c_str(), wxConvUTF8);
    if (!c->dmz()->CheckDeviceInputs(devs[i].d))
      desc = desc + wxT(" ") + _("(disconnected input)");
    devNames.Add(desc);
  }
  Set(devNames);

  if (selectedIndex<0)
  {
    if (devs.size()!=0)
      selectedDev->Set(devs[0].d);
    else
      selectedDev->Set(NULL);
  }
  else
  {
    SetSelection(selectedIndex);
  }
}

void DevicesListBox::OnDeviceSelectionChanged()
{
  if (!selectedDev || isSetting) return;

  for (int i=0; i<devs.size(); i++)
  {
    if (devs[i].d = selectedDev->Get())
    {
```

```
      SetSelection(i);
      return;
    }
  }
  SetSelection(wxNOT_FOUND);
}

void DevicesListBox::OnLBoxSelectionChanged(wxCommandEvent& event)
{
  if (!selectedDev) return;

  devlink d;
  int i = GetSelection();
  if (i>=0 && i<devs.size())
  {
    isSetting = true;// temporarily prevent DevicesListBox::
    OnDeviceSelectionChanged() from being run, otherwise the
    selection doesn't change properly
    selectedDev->Set(devs[i].d);
    isSetting = false;
  }
}

devlink DevicesListBox::GetSelectionAfterDelete()
{
  if (!c || !selectedDev || !selectedDev->Get())
  {
    if (devs.size()) return devs[0].d;
    else return NULL;
  }
  if (devs.size()<=1) return NULL;

  int i = GetSelection()+1;
  if (i>=devs.size()) i = devs.size()-1;
  if (i==GetSelection()) i = 0;
  return devs[i].d;
}

BEGIN_EVENT_TABLE(DevicesListBox, wxListBox)
  EVT_LISTBOX(wxID_ANY, DevicesListBox::OnLBoxSelectionChanged)
END_EVENT_TABLE()


void ShowErrorMsgDialog(wxWindow* parent, wxString txt)
{
  wxMessageDialog dlg(parent, txt, _("Error"), wxCANCEL |
    wxICON_ERROR);
  dlg.ShowModal();
}

```

```
void CircuitElementInfoVector_to_wxArrayString(const
    CircuitElementInfoVector& src, wxArrayString& dest)
{
  dest.Clear();
  dest.Alloc(src.size());
  for (CircuitElementInfoVector::const_iterator it=src.begin(); it<
    src.end(); ++it)
  {
    dest.Add(wxString(it->namestr.c_str(), wxConvUTF8));
  }
}
```

3.2.6 GUI: widget IDs

Listing 18: gui-id.h

```
#ifndef gui_id_h
#define gui_id_h

enum
{
    MY_SPINCNTRL_ID = wxID_HIGHEST + 1,
    MY_TEXTCTRL_ID,
    MY_BUTTON_ID,
    MENU_CLEAR_CIRCUIT,
    OUTPUT_TEXTCTRL_ID,
    SIMCTRL_BUTTON_RUN_ID,
    SIMCTRL_BUTTON_CONT_ID,
    MONITORS_ADD_BUTTON_ID,
    MONITORS_DEL_BUTTON_ID,
    SWITCHES_CTRL_ID,
    DEVICES_EDIT_BUTTON_ID,
    DEVICES_APPLY_BUTTON_ID,
    DEVICECREATE_BUTTON_ID,
    DEVICEDELETE_BUTTON_ID,
    DEVICES_ADDCONN_BUTTON_ID,
    DEVICES_DELCONN_BUTTON_ID,
    DEVICENAME_TEXTCTRL_ID,
    DEVICEOUTPUT_MONITOR_CB_ID
};

#endif /* gui_id_h */
```

3.2.7 Device editing GUI: dialogs

Listing 19: gui-devices.h

```
#ifndef gui_devices_h
#define gui_devices_h

#include "gui-devices-infopanels.h"
#include <wx/wx.h>
#include <wx/spinctrl.h>
#include <wx/listbox.h>
#include <wx/panel.h>
#include <wx/combobox.h>
#include "network.h"
#include "circuit.h"
#include "observer.h"
#include "gui-misc.h"
#include <string>
#include <vector>
using namespace std;


class NewDeviceDialog: public wxDialog
{
public:
  NewDeviceDialog(circuit* circ, wxWindow* parent, wxWindowID id,
    const wxString& title);
  devlink newdev;
private:
  circuit* c;
  DevicekindDropdown* dkindDropdown;
  DeviceNameTextCtrl* devicenameCtrl;
  void OnOK(wxCommandEvent& event);
  DECLARE_EVENT_TABLE()
};

class ChooseOutputDialog: public wxDialog
{
public:
  ChooseOutputDialog(circuit* circ, wxWindow* parent, wxWindowID id,
    const wxString& title, const wxString& description);
  CircuitElementInfo result;
private:
  circuit* c;
  wxListBox* lbox;
  CircuitElementInfoVector outputs;
  void OnOK(wxCommandEvent& event);
  DECLARE_EVENT_TABLE()
};

// For a given output, allow input(s) to be chosen to connect to it
// Makes the connection(s) when the OK button is clicked
class ConnectToOutputDialog: public wxDialog
{
public:
  ConnectToOutputDialog(circuit* circ, outplink outp, wxWindow*
    parent, wxWindowID id, const wxString& title);
private:
  circuit* c;
  outplink o;
  CircuitElementInfoVector inputs;
  wxListBox* lbox;
  void OnOK(wxCommandEvent& event);
  DECLARE_EVENT_TABLE()
};

class DevicesDialog: public wxDialog
{
public:
  DevicesDialog(circuit* circ, wxWindow* parent, wxWindowID id, const
     wxString& title, devlink d=NULL);
  ~DevicesDialog();
  void OnDeviceSelectionChanged();
private:
  circuit* c;
  SelectedDevice* selectedDev;
  DevicesListBox* devListBox;
  DeviceDetailsPanel* detailsPanel;
  DeviceInputsPanel* inputsPanel;
  vector<DeviceOutputPanel*> outputPanels;
  wxBoxSizer* mainSizer;
  wxBoxSizer* outputsSizer;
  void DestroyDeviceWidgets();
  void OnDeleteButton(wxCommandEvent& event);
  void OnCreateButton(wxCommandEvent& event);
  DECLARE_EVENT_TABLE()
};



#endif /* gui_devices_h */
```

Listing 20: gui-devices.cc

```
#include "gui-devices.h"
#include "gui-id.h"
#include "network.h"
#include "wx_icon.xpm"
#include <algorithm>
#include <climits>
using namespace std;

// The devices editing GUI
```

```
DevicesDialog::DevicesDialog(circuit* circ, wxWindow* parent,
    wxWindowID id, const wxString& title, devlink d) :
  wxDialog(parent, id, title, wxDefaultPosition, wxDefaultSize,
    wxDEFAULT_DIALOG_STYLE | wxRESIZE_BORDER)
{
  c = circ;
  outputsSizer = NULL;
  outputPanels.resize(0);
  inputsPanel = NULL;
  detailsPanel = NULL;
  // A class to store a pointer to the currently selected device, and
     an ObserverSubject which is triggered when it changes
  selectedDev = new SelectedDevice();
  selectedDev->Set(d);
  selectedDev->changed.Attach(this, &DevicesDialog::
    OnDeviceSelectionChanged);

  wxBoxSizer* topsizer = new wxBoxSizer(wxHORIZONTAL);
  wxBoxSizer* deviceListSizer = new wxBoxSizer(wxVERTICAL);
  wxBoxSizer* ioSizer = new wxBoxSizer(wxHORIZONTAL);
  mainSizer = new wxBoxSizer(wxVERTICAL);
  // List of devices, which can be used to change the selected device
  devListBox = new DevicesListBox(c, selectedDev, this, wxID_ANY);
  devListBox->SetMinSize(wxSize(150,-1));
  deviceListSizer->Add(devListBox, 1, wxALL | wxEXPAND, 10);
  deviceListSizer->Add(new wxButton(this, DEVICECREATE_BUTTON_ID, _("
    Add device")), 0, (wxALL & ~wxTOP) | wxEXPAND, 10);

  // Widgets to display device details
  inputsPanel = new DeviceInputsPanel(c, selectedDev, this);
  ioSizer->Add(inputsPanel, 1, wxEXPAND | wxALL, 5);
  outputsSizer = new wxBoxSizer(wxVERTICAL);
  ioSizer->Add(outputsSizer, 1, wxEXPAND | wxALL, 0);
  mainSizer->Add(ioSizer, 1, wxEXPAND | wxALL, 5);

  topsizer->Add(deviceListSizer, 0, wxEXPAND, 0);
  topsizer->Add(mainSizer, 3, wxEXPAND, 0);

  mainSizer->Add(CreateButtonSizer(wxOK), 0, wxALL | wxEXPAND, 10);
  SetSizerAndFit(topsizer);
  SetSizeHints(400,300);

  OnDeviceSelectionChanged();
}

DevicesDialog::~DevicesDialog()
{
  // detach all widgets from selectedDev->changed observer before
    deleting selectedDev
  selectedDev->changed.Detach(this);
  devListBox->ReleasePointers();
  inputsPanel->ReleasePointers();
  for (vector<DeviceOutputPanel*>::iterator it=outputPanels.begin();
    it<outputPanels.end(); ++it)
  {
    (*it)->ReleasePointers();
  }
  delete selectedDev;
}

void DevicesDialog::OnDeviceSelectionChanged()
{
  // The widgets in the device details panel and the number of output
     details panels vary according to devicekind, so it's easier to
    just remove them all and recreate when a different device is
    selected

  DestroyDeviceWidgets();

  if (!selectedDev || !c || !outputsSizer) return;

  devlink d = selectedDev->Get();

  // Device details panel
  detailsPanel = new DeviceDetailsPanel(c, selectedDev, this);
  mainSizer->Insert(0, detailsPanel, 0, wxEXPAND | wxALL, 10);

  if (selectedDev->Get())
  {
    // Make a sorted list of device outputs
    CircuitElementInfoVector outputs;
    outputs.push_back_dev_outputs(selectedDev->Get());
    outputs.UpdateSignalNames(c);
    sort(outputs.begin(), outputs.end(),
    CircuitElementInfo_namestrcmp);
    // Create a DeviceOutputPanel for each
    for (CircuitElementInfoVector::iterator it=outputs.begin(); it<
    outputs.end(); ++it)
    {
      DeviceOutputPanel* opanel = new DeviceOutputPanel(c, it->o,
    this);
      outputPanels.push_back(opanel);
      outputsSizer->Add(opanel, 1, wxEXPAND | wxALL, 5);
    }
  }

  Layout();
  Fit();
}

void DevicesDialog::DestroyDeviceWidgets()
{
```

```
  // Remove the device details and output details panels
  for (vector<DeviceOutputPanel*>::iterator it=outputPanels.begin();
    it<outputPanels.end(); ++it)
  {
    (*it)->Destroy();
  }
  outputPanels.resize(0);
  if (detailsPanel)
  {
    detailsPanel->Destroy();
    detailsPanel = NULL;
  }
}

void DevicesDialog::OnDeleteButton(wxCommandEvent& event)
{
  if (!c || !selectedDev || !selectedDev->Get()) return;

  devlink d = selectedDev->Get();
  // Get the device which should be selected after the currently
    selected one is deleted (usually the one after it in the list),
    and select it before deleting the currently selected device
  devlink newDev = devListBox->GetSelectionAfterDelete();
  selectedDev->Set(newDev);
  // Delete device (and related monitors)
  c->RemoveDevice(d);
}

void DevicesDialog::OnCreateButton(wxCommandEvent& event)
{
  if (!c || !selectedDev) return;
  // Dialog to select new device type
  NewDeviceDialog dlg(c, this, wxID_ANY, _("Create new device"));
  if (dlg.ShowModal()==wxID_OK && dlg.newdev)
  {
    // Select the new device (created by dialog when OK is clicked)
    selectedDev->Set(dlg.newdev);
    c->circuitChanged.Trigger();
  }
}

BEGIN_EVENT_TABLE(DevicesDialog, wxDialog)
  EVT_BUTTON(DEVICEDELETE_BUTTON_ID, DevicesDialog::OnDeleteButton)
  EVT_BUTTON(DEVICECREATE_BUTTON_ID, DevicesDialog::OnCreateButton)
END_EVENT_TABLE()


ChooseOutputDialog::ChooseOutputDialog(circuit* circ, wxWindow*
    parent, wxWindowID id, const wxString& title, const wxString&
    description):
  wxDialog(parent, id, title, wxDefaultPosition, wxDefaultSize,
    wxDEFAULT_DIALOG_STYLE | wxRESIZE_BORDER)
{
  SetIcon(wxIcon(wx_icon));

  c = circ;

  // Make a list of all outputs in the circuit, for the listbox
  outputs.push_back_all_outputs(c->netz()->devicelist());
  outputs.UpdateSignalNames(c);
  sort(outputs.begin(), outputs.end(), CircuitElementInfo_namestrcmp)
    ;

  wxArrayString displayedOutputs;
  CircuitElementInfoVector_to_wxArrayString(outputs, displayedOutputs
    );

  // Create widgets
  wxBoxSizer* topsizer = new wxBoxSizer(wxVERTICAL);
  topsizer->Add(new wxStaticText(this, wxID_ANY, description), 0,
    wxLEFT | wxRIGHT | wxTOP, 10);
  lbox = new wxListBox(this, wxID_ANY, wxDefaultPosition,
    wxDefaultSize, displayedOutputs, wxLB_SINGLE | wxLB_NEEDED_SB);
  topsizer->Add(lbox, 1, wxALL | wxEXPAND, 10);
  topsizer->Add(CreateButtonSizer(wxOK | wxCANCEL), 0, wxALL |
    wxEXPAND, 10);
  SetSizerAndFit(topsizer);
}

void ChooseOutputDialog::OnOK(wxCommandEvent& event)
{
  int i = lbox->GetSelection();
  if (i>=0 && i<outputs.size())
  {
    // Store a pointer to the selected output in a public member
    variable
    result = outputs[i];
  }
  EndModal(wxID_OK);
}

BEGIN_EVENT_TABLE(ChooseOutputDialog, wxDialog)
  EVT_BUTTON(wxID_OK, ChooseOutputDialog::OnOK)
END_EVENT_TABLE()


ConnectToOutputDialog::ConnectToOutputDialog(circuit* circ, outplink
    outp, wxWindow* parent, wxWindowID id, const wxString& title):
  wxDialog(parent, id, title, wxDefaultPosition, wxDefaultSize,
    wxDEFAULT_DIALOG_STYLE | wxRESIZE_BORDER)
{
```

```
  SetIcon(wxIcon(wx_icon));

  c = circ;
  o = outp;

  // Make a list of all inputs in the circuit, for the listbox
  inputs.push_back_all_inputs(c->netz()->devicelist());
  inputs.UpdateSignalNames(c);
  // Sort by whether the input is connected, then alphabetically by name
  sort(inputs.begin(), inputs.end(), CircuitElementInfo_iconnect_namestrcmp);

  // Put the input names in the listbox, with a description of what (if anything) is connected to each one
  wxArrayString lboxContents;
  lboxContents.Alloc(inputs.size());
  for (CircuitElementInfoVector::iterator it=inputs.begin(); it<inputs.end(); ++it)
  {
    wxString desc(it->namestr.c_str(), wxConvUTF8);
    if (it->i->connect)
    {
      desc = desc + wxT(" (") + _("connected to ") + wxString(c->netz()->getsignalstring(it->i->connect->dev->id, it->i->connect->id).c_str(), wxConvUTF8) + wxT(")");
    }
    else
    {
      desc = desc + _(" (unconnected)");
    }
    lboxContents.Add(desc);
  }

  // Create widgets
  wxBoxSizer* topsizer = new wxBoxSizer(wxVERTICAL);
  topsizer->Add(new wxStaticText(this, wxID_ANY, wxString(_("Choose input(s) to connect to ")) + wxString(c->netz()->getsignalstring(o->dev, o).c_str(), wxConvUTF8)), 0, wxLEFT | wxRIGHT | wxTOP, 10);
  lbox = new wxListBox(this, wxID_ANY, wxDefaultPosition, wxDefaultSize, lboxContents, wxLB_EXTENDED | wxLB_NEEDED_SB);
  topsizer->Add(lbox, 1, wxALL | wxEXPAND, 10);
  topsizer->Add(CreateButtonSizer(wxOK | wxCANCEL), 0, wxALL | wxEXPAND, 10);
  SetSizerAndFit(topsizer);
}

void ConnectToOutputDialog::OnOK(wxCommandEvent& event)
{
  // Connect all the selected inputs to the output that was passed in the constructor
  wxArrayInt selections;
  lbox->GetSelections(selections);
  for (int i=0; i<selections.GetCount(); i++)
  {
    if (selections[i]<inputs.size())
      inputs[selections[i]].i->connect = o;
  }
  c->circuitChanged.Trigger();
  EndModal(wxID_OK);
}

BEGIN_EVENT_TABLE(ConnectToOutputDialog, wxDialog)
  EVT_BUTTON(wxID_OK, ConnectToOutputDialog::OnOK)
END_EVENT_TABLE()


NewDeviceDialog::NewDeviceDialog(circuit* circ,wxWindow* parent, wxWindowID id, const wxString& title):
  wxDialog(parent, id, title, wxDefaultPosition, wxDefaultSize, wxDEFAULT_DIALOG_STYLE | wxRESIZE_BORDER)
{
  SetIcon(wxIcon(wx_icon));

  c = circ;

  wxBoxSizer* topsizer = new wxBoxSizer(wxVERTICAL);
  // Device name textbox
  topsizer->Add(new wxStaticText(this, wxID_ANY, _("New device name:")), 0, wxLEFT | wxRIGHT | wxTOP, 10);
  devicenameCtrl = new DeviceNameTextCtrl(this);
  topsizer->Add(devicenameCtrl, 0, wxALL | wxEXPAND, 10);
  // Device type dropdown
  topsizer->Add(new wxStaticText(this, wxID_ANY, _("Device type:")), 0, wxLEFT | wxRIGHT | wxTOP, 10);
  dkindDropdown = new DevicekindDropdown(this);
  dkindDropdown->SetSelection(0);// Select a default device type
  topsizer->Add(dkindDropdown, 0, wxALL | wxEXPAND, 10);
  topsizer->Add(CreateButtonSizer(wxOK | wxCANCEL), 0, wxALL | wxEXPAND, 10);
  SetSizerAndFit(topsizer);
}

void NewDeviceDialog::OnOK(wxCommandEvent& event)
{
  if (!devicenameCtrl->CheckValid(c, blankname, true))
    return;

  // Create a new device of the specified type, with some defaults for any options (like gate input count)
```

```
  // Options can be changed immediately after creation in the device
    editing GUI
  devicekind dk = dkindDropdown->GetDevicekind();
  name devname = c->nmz()->lookup(string(devicenameCtrl->GetValue().
    mb_str()));
  bool ok;
  if (dk==aswitch)
    c->dmz()->makedevice(dk, devname, 1, ok);
  else if (dk==aclock)
    c->dmz()->makedevice(dk, devname, 5, ok);
  else
    c->dmz()->makedevice(dk, devname, 2, ok);

  // Put a pointer to the newly created device in a public member
    variable
  newdev = c->netz()->finddevice(devname);
  EndModal(wxID_OK);
}

BEGIN_EVENT_TABLE(NewDeviceDialog, wxDialog)
  EVT_BUTTON(wxID_OK, NewDeviceDialog::OnOK)
END_EVENT_TABLE()
```

3.2.8 Device editing GUI: information panels

Listing 21: gui-devices-infopanels.h

```
#ifndef gui_devices_infopanels_h
#define gui_devices_infopanels_h

#include "circuit.h"
#include "observer.h"
#include "gui-misc.h"
#include <wx/panel.h>
#include <wx/gbsizer.h>
#include <wx/combobox.h>
#include <wx/textctrl.h>
#include <wx/stattext.h>
#include <wx/spinctrl.h>
#include <wx/checkbox.h>
#include <wx/button.h>

// Panel for display and modification of device properties
class DeviceDetailsPanel: public wxPanel
{
public:
  DeviceDetailsPanel(circuit* circ, SelectedDevice* selectedDev_in,
    wxWindow* parent, wxWindowID id = wxID_ANY);
protected:
  virtual void UpdateApplyButtonState_ExtraFields() {};
  virtual void OnApply_ExtraFields(bool& changedSomething) {};
  SelectedDevice* selectedDev;
  circuit* c;
  wxStaticBoxSizer* mainSizer;
  wxGridBagSizer* gridsizer;
  DeviceNameTextCtrl* devicenameCtrl;
  wxButton* updateBtn;
  wxStaticText* devicekindStaticText;

  wxSpinCtrl* spinCtrl;
  DevicekindDropdown* gateTypeDropdown;

  void UpdateApplyButtonState();
  void OnInputChanged(wxCommandEvent& event);
  void OnApply(wxCommandEvent& event);
  void ShowErrorMsg(wxString txt);
  DECLARE_EVENT_TABLE()
};

// List of all inputs on a device and which output each is connected
    to, with buttons to add/remove connections
class DeviceInputsPanel: public wxPanel
{
public:
  DeviceInputsPanel(circuit* circ, SelectedDevice* selectedDev_in,
    wxWindow* parent, wxWindowID id = wxID_ANY);
  ~DeviceInputsPanel();
  void ReleasePointers();
private:
  SelectedDevice* selectedDev;
  circuit* c;
  CircuitElementInfoVector inps;
  wxListBox* lbox;
  wxButton *btnAddConn, *btnDelConn;
  void UpdateInps();
  void OnConnectButton(wxCommandEvent& event);
  void OnDisconnectButton(wxCommandEvent& event);
  void OnLBoxSelectionChanged(wxCommandEvent& event);
  void UpdateControlStates();
  void OnDeviceSelectionChanged();
  DECLARE_EVENT_TABLE()
};

// Lists the inputs of other devices that a particular device output
    is connected to, with buttons to add/remove connections
class DeviceOutputPanel: public wxPanel
{
public:
```

```
  DeviceOutputPanel(circuit* circ, outplink targetOutp, wxWindow*
    parent, wxWindowID id = wxID_ANY);
  ~DeviceOutputPanel();
  void ReleasePointers();
private:
  circuit* c;
  outplink outp;
  CircuitElementInfoVector inps;
  wxListBox* lbox;
  wxButton *btnAddConn, *btnDelConn;
  wxCheckBox* monitorCheckbox;
  void UpdateInps();
  void OnConnectButton(wxCommandEvent& event);
  void OnDisconnectButton(wxCommandEvent& event);
  void OnLBoxSelectionChanged(wxCommandEvent& event);
  void OnMonitorCheckboxChanged(wxCommandEvent& event);
  void UpdateControlStates();
  void OnDeviceSelectionChanged();
  void OnMonitorsChanged();
  DECLARE_EVENT_TABLE()
};

#endif /* gui_devices_infopanels_h */
```

Listing 22: gui-devices-infopanels.cc

```
#include "gui-devices-infopanels.h"
#include "gui-devices.h"
#include "gui-id.h"
#include <algorithm>

using namespace std;

DeviceInputsPanel::DeviceInputsPanel(circuit* circ, SelectedDevice*
    selectedDev_in, wxWindow* parent, wxWindowID id) :
  wxPanel(parent, id)
{
  selectedDev = selectedDev_in;
  c = circ;

  wxStaticBoxSizer* mainSizer = new wxStaticBoxSizer(wxVERTICAL, this
    , _("Inputs"));
  // Listbox to show all inputs for this device
  lbox = new wxListBox(this, wxID_ANY, wxDefaultPosition,
    wxDefaultSize, 0, NULL, wxLB_NEEDED_SB | wxLB_EXTENDED);
  lbox->SetMinSize(wxSize(150,100));
  mainSizer->Add(lbox, 1, wxEXPAND | (wxALL & ~wxBOTTOM), 10);
  // Connect and disconnect buttons
  wxBoxSizer* buttonsSizer = new wxBoxSizer(wxHORIZONTAL);
  btnAddConn = new wxButton(this, DEVICES_ADDCONN_BUTTON_ID, _("
    Connect"));
  btnDelConn = new wxButton(this, DEVICES_DELCONN_BUTTON_ID, _("
    Disconnect"));
  buttonsSizer->Add(btnAddConn, 1, wxEXPAND, 10);
  buttonsSizer->Add(btnDelConn, 1, wxEXPAND, 10);
  mainSizer->Add(buttonsSizer, 0, wxEXPAND | (wxALL & ~wxTOP) , 10);
  SetSizerAndFit(mainSizer);

  // Listen for device selection changes, and changes to device
    details
  selectedDev->changed.Attach(this, &DeviceInputsPanel::
    OnDeviceSelectionChanged);
  c->circuitChanged.Attach(this, &DeviceInputsPanel::UpdateInps);

  UpdateInps();
  UpdateControlStates();
}

DeviceInputsPanel::~DeviceInputsPanel()
{
  ReleasePointers();
}

void DeviceInputsPanel::ReleasePointers()
{
  if (selectedDev)
    selectedDev->changed.Detach(this);
  selectedDev = NULL;
  if (c)
    c->circuitChanged.Detach(this);
  c = NULL;
}

void DeviceInputsPanel::UpdateInps()
{
  wxArrayInt selections;
  lbox->GetSelections(selections);

  inps.resize(0);
  wxArrayString lboxData;
  if (selectedDev && c && selectedDev->Get())
  {
    // Make a list of all inputs for the selected device
    inps.push_back_dev_inputs(selectedDev->Get());
    // Reverse it, since creating from I1..I16 means ilist points to
    the last one, I16
    reverse(inps.begin(), inps.end());
    // Put input names in the listbox, with a description of what (if
     anything) is connected to each one
    lboxData.Alloc(inps.size());
    for (CircuitElementInfoVector::iterator it=inps.begin(); it<inps.
    end(); ++it)
```

```
    {
      inplink i = it->i;
      wxString desc(c->nmz()->getnamestring(i->id).c_str(),
    wxConvUTF8);
      if (i->connect)
      {
        desc = desc + wxT(" (") + _("connected to ") + wxString(c->
    netz()->getsignalstring(i->connect->dev->id, i->connect->id).
    c_str(), wxConvUTF8) + wxT(")");
      }
      else
      {
        desc = desc + _(" (unconnected)");
      }
      lboxData.Add(desc);
    }
  }
  lbox->Set(lboxData);

  UpdateControlStates();
}

void DeviceInputsPanel::OnDeviceSelectionChanged()
{
  lbox->SetSelection(wxNOT_FOUND);
  if (c && selectedDev && selectedDev->Get())
  {
    // Selected device changed, update the displayed inputs
    UpdateInps();
    Show();
  }
  else
  {
    // Hide the device inputs details panel if no device is selected
    Hide();
  }
}

void DeviceInputsPanel::OnLBoxSelectionChanged(wxCommandEvent& event)
{
  UpdateControlStates();
}

void DeviceInputsPanel::UpdateControlStates()
{
  // Enable or disable add+remove connection buttons depending on
    whether any inputs are selected
  wxArrayInt selections;
  lbox->GetSelections(selections);
  if (selections.GetCount())
  {
    btnAddConn->Enable();
    btnDelConn->Enable();
  }
  else
  {
    btnAddConn->Disable();
    btnDelConn->Disable();
  }
}

void DeviceInputsPanel::OnConnectButton(wxCommandEvent& event)
{
  wxArrayInt selections;
  lbox->GetSelections(selections);
  ChooseOutputDialog dlg(c, this, wxID_ANY, _("Choose output"),
    wxPLURAL("Choose an output to connect this input to:", "Choose an
     output to connect these inputs to:", selections.GetCount()));
  if (dlg.ShowModal()==wxID_OK && dlg.result.o)
  {
    wxArrayInt selections;
    lbox->GetSelections(selections);
    // Connect all selected inputs to the output chosen in the dialog
    for (int i=0; i<selections.GetCount(); i++)
    {
      if (selections[i]<inps.size())
      {
        inps[selections[i]].i->connect = dlg.result.o;
      }
    }
    c->circuitChanged.Trigger();
  }
}

void DeviceInputsPanel::OnDisconnectButton(wxCommandEvent& event)
{
  wxArrayInt selections;
  lbox->GetSelections(selections);
  // Disconnect all selected inputs from whatever they are connected
    to
  for (int i=0; i<selections.GetCount(); i++)
  {
    if (i<inps.size()) inps[selections[i]].i->connect = NULL;
  }
  c->circuitChanged.Trigger();
}

BEGIN_EVENT_TABLE(DeviceInputsPanel, wxPanel)
  EVT_LISTBOX(wxID_ANY, DeviceInputsPanel::OnLBoxSelectionChanged)
  EVT_BUTTON(DEVICES_ADDCONN_BUTTON_ID, DeviceInputsPanel::
    OnConnectButton)
```

```
  EVT_BUTTON(DEVICES_DELCONN_BUTTON_ID, DeviceInputsPanel::
    OnDisconnectButton)
END_EVENT_TABLE()


DeviceOutputPanel::DeviceOutputPanel(circuit* circ, outplink
    targetOutp, wxWindow* parent, wxWindowID id) :
  wxPanel(parent, id)
{
  outp = targetOutp;
  c = circ;

  wxString title = _("Output");
  if (outp->id != blankname)
    title = title + wxT(" ") + wxString(c->nmz()->getnamestring(outp
    ->id).c_str(),wxConvUTF8);
  wxStaticBoxSizer* mainSizer = new wxStaticBoxSizer(wxVERTICAL, this
    , title);
  // Checkbox to add/remove a monitor on this output
  monitorCheckbox = new wxCheckBox(this, DEVICEOUTPUT_MONITOR_CB_ID,
    _("Monitored"));
  mainSizer->Add(monitorCheckbox, 0, wxALL, 10);
  // Listbox showing all connected inputs
  mainSizer->Add(new wxStaticText(this, wxID_ANY, _("Inputs connected
     to this output:")), 0, wxLEFT | wxRIGHT, 10);
  lbox = new wxListBox(this, wxID_ANY, wxDefaultPosition,
    wxDefaultSize, 0, NULL, wxLB_NEEDED_SB | wxLB_EXTENDED);
  lbox->SetMinSize(wxSize(150,50));
  mainSizer->Add(lbox, 1, wxEXPAND | wxLEFT | wxRIGHT, 10);
  // Connect and disconnect buttons
  wxBoxSizer* buttonsSizer = new wxBoxSizer(wxHORIZONTAL);
  btnAddConn = new wxButton(this, DEVICES_ADDCONN_BUTTON_ID, _("
    Connect"));
  btnDelConn = new wxButton(this, DEVICES_DELCONN_BUTTON_ID, _("
    Disconnect"));
  buttonsSizer->Add(btnAddConn, 1, wxEXPAND, 10);
  buttonsSizer->Add(btnDelConn, 1, wxEXPAND, 10);
  mainSizer->Add(buttonsSizer, 0, wxEXPAND | (wxALL & ~wxTOP) , 10);
  SetSizerAndFit(mainSizer);

  // Listen for changes to device details and monitors
  c->circuitChanged.Attach(this, &DeviceOutputPanel::UpdateInps);
  c->monitorsChanged.Attach(this, &DeviceOutputPanel::
    OnMonitorsChanged);

  UpdateInps();
  UpdateControlStates();
  OnMonitorsChanged();
}

DeviceOutputPanel::~DeviceOutputPanel()
{
  ReleasePointers();
}

void DeviceOutputPanel::ReleasePointers()
{
  if (c)
  {
    c->circuitChanged.Detach(this);
    c->monitorsChanged.Detach(this);
  }
  c = NULL;
}

void DeviceOutputPanel::OnMonitorsChanged()
{
  // If monitors are added or removed, update the value of the
    checkbox indicating whether this output is monitored
  monitorCheckbox->SetValue(c->mmz()->IsMonitored(outp));
}

void DeviceOutputPanel::OnMonitorCheckboxChanged(wxCommandEvent&
    event)
{
  bool ok = false;
  // Monitor checkbox changed, add or remove a monitor for this
    device
  if (monitorCheckbox->IsChecked())
  {
    c->mmz()->makemonitor(outp->dev->id, outp->id, ok);
    if (c->GetTotalCycles()!=0)
      cout << wxString(_("Monitor added, run simulation again to see
    updated signals")).mb_str() << endl;
  }
  else
  {
    c->mmz()->remmonitor(outp->dev->id, outp->id, ok);
  }
  if (ok) c->monitorsChanged.Trigger();
}

void DeviceOutputPanel::UpdateInps()
{
  wxArrayInt selections;
  lbox->GetSelections(selections);
  inps.resize(0);
  wxArrayString lboxData;
  if (c)
  {
    // Make a list of all inputs connected to this output
    devlink d = c->netz()->devicelist();
```

```
    while (d != NULL)
    {
      inplink i = d->ilist;
      while (i != NULL)
      {
        if (i->connect == outp) inps.push_back(CircuitElementInfo(d,i
    ));
        i = i->next;
      }
      d = d->next;
    }
    inps.UpdateSignalNames(c);
    sort(inps.begin(), inps.end(), CircuitElementInfo_namestrcmp);
    // Put the names in the listbox
    lboxData.Alloc(inps.size());
    for (CircuitElementInfoVector::iterator it=inps.begin(); it<inps.
    end(); ++it)
    {
      lboxData.Add(wxString(it->namestr.c_str(), wxConvUTF8));
    }
  }
  lbox->Set(lboxData);
  UpdateControlStates();
}

void DeviceOutputPanel::OnLBoxSelectionChanged(wxCommandEvent& event)
{
  UpdateControlStates();
}

void DeviceOutputPanel::UpdateControlStates()
{
  // Enable or disable "remove connection" button depending on
    whether any inputs are currently selected
  wxArrayInt selections;
  lbox->GetSelections(selections);
  if (selections.GetCount())
  {
    btnDelConn->Enable();
  }
  else
  {
    btnDelConn->Disable();
  }
}

void DeviceOutputPanel::OnConnectButton(wxCommandEvent& event)
{
  // Dialog to choose some inputs to connect to this output
  ConnectToOutputDialog dlg(c, outp, this, wxID_ANY, _("Choose inputs
    "));
  dlg.ShowModal();
}

void DeviceOutputPanel::OnDisconnectButton(wxCommandEvent& event)
{
  // Disconnect all selected inputs
  wxArrayInt selections;
  lbox->GetSelections(selections);
  for (int i=0; i<selections.GetCount(); i++)
  {
    if (i<inps.size()) inps[selections[i]].i->connect = NULL;
  }
  c->circuitChanged.Trigger();
}

BEGIN_EVENT_TABLE(DeviceOutputPanel, wxPanel)
  EVT_LISTBOX(wxID_ANY, DeviceOutputPanel::OnLBoxSelectionChanged)
  EVT_CHECKBOX(DEVICEOUTPUT_MONITOR_CB_ID, DeviceOutputPanel::
    OnMonitorCheckboxChanged)
  EVT_BUTTON(DEVICES_ADDCONN_BUTTON_ID, DeviceOutputPanel::
    OnConnectButton)
  EVT_BUTTON(DEVICES_DELCONN_BUTTON_ID, DeviceOutputPanel::
    OnDisconnectButton)
END_EVENT_TABLE()


DeviceDetailsPanel::DeviceDetailsPanel(circuit* circ, SelectedDevice*
     selectedDev_in, wxWindow* parent, wxWindowID id) :
  wxPanel(parent, id)
{
  selectedDev = selectedDev_in;
  c = circ;

  mainSizer = new wxStaticBoxSizer(wxVERTICAL, this, _("Device
    details"));
  devicekind dk = baddevice;
  devlink d = selectedDev->Get();
  if (d)
  {
    dk = d->kind;

    wxBoxSizer* buttonsSizer = new wxBoxSizer(wxHORIZONTAL);
    gridsizer = new wxGridBagSizer();
    wxString kindtext = wxT("Unknown device");
    if (dk>=0 && dk<baddevice)
      kindtext = devicekindstrings[dk];

    // Device type text
    gridsizer->Add(new wxStaticText(this, wxID_ANY, _("Device type:")
    ), wxGBPosition(0,0), wxDefaultSpan, (wxALL & ~wxRIGHT) |
    wxALIGN_CENTER_VERTICAL, 10);
```

```
    devicekindStaticText = new wxStaticText(this, wxID_ANY, kindtext);
    gridsizer->Add(devicekindStaticText, wxGBPosition(0,1), wxDefaultSpan, wxALL | wxALIGN_CENTER_VERTICAL, 10);
    // Device name textbox
    gridsizer->Add(new wxStaticText(this, wxID_ANY, _("Name:")), wxGBPosition(0,3), wxDefaultSpan, (wxALL & ~wxRIGHT) | wxALIGN_CENTER_VERTICAL, 10);
    devicenameCtrl = new DeviceNameTextCtrl(this, DEVICENAME_TEXTCTRL_ID, wxString(c->nmz()->getnamestring(d->id).c_str(), wxConvUTF8));
    gridsizer->Add(devicenameCtrl, wxGBPosition(0,4), wxDefaultSpan, wxALL | wxEXPAND | wxALIGN_CENTER_VERTICAL, 10);
    gridsizer->AddGrowableCol(1,3);
    gridsizer->AddGrowableCol(4,5);

    // Widgets specific to particular devices
    if (d->kind == aclock)
    {
      // Half time period ("frequency" in existing code)
      spinCtrl = new wxSpinCtrl(this);
      spinCtrl->SetValue(selectedDev->Get()->frequency);
      spinCtrl->SetRange(1,INT_MAX);
      gridsizer->Add(new wxStaticText(this, wxID_ANY, _("Half-time-period:")), wxGBPosition(1,3), wxDefaultSpan, wxLEFT | wxBOTTOM | wxALIGN_CENTER_VERTICAL, 10);
      gridsizer->Add(spinCtrl, wxGBPosition(1,4), wxDefaultSpan, (wxALL & ~wxTOP) | wxEXPAND | wxALIGN_CENTER_VERTICAL, 10);
    }
    else if (d->kind == andgate || d->kind == nandgate || d->kind == orgate || d->kind == norgate)
    {
      // Gate type dropdown (allow changes of device kind between and/nand/or/nor, since they have identical parameters and input/output rules)
      vector<devicekind> gateTypeChoices;
      gateTypeChoices.push_back(andgate);
      gateTypeChoices.push_back(nandgate);
      gateTypeChoices.push_back(orgate);
      gateTypeChoices.push_back(norgate);
      gateTypeDropdown = new DevicekindDropdown(this, wxID_ANY, gateTypeChoices);
      gateTypeDropdown->SetDevicekind(d->kind);
      gridsizer->Add(new wxStaticText(this, wxID_ANY, _("Gate type:")), wxGBPosition(1,0), wxDefaultSpan, wxLEFT | wxBOTTOM | wxALIGN_CENTER_VERTICAL, 10);
      gridsizer->Add(gateTypeDropdown, wxGBPosition(1,1), wxDefaultSpan, (wxALL & ~wxTOP) | wxEXPAND | wxALIGN_CENTER_VERTICAL, 10);
      // Spinner to select number of inputs
      spinCtrl = new wxSpinCtrl(this);
      spinCtrl->SetValue(GetLinkedListLength(selectedDev->Get()->ilist));
      spinCtrl->SetRange(1,16);
      gridsizer->Add(new wxStaticText(this, wxID_ANY, _("Inputs:")), wxGBPosition(1,3), wxDefaultSpan, wxLEFT | wxBOTTOM | wxALIGN_CENTER_VERTICAL, 10);
      gridsizer->Add(spinCtrl, wxGBPosition(1,4), wxDefaultSpan, (wxALL & ~wxTOP) | wxEXPAND | wxALIGN_CENTER_VERTICAL, 10);
    }
    mainSizer->Add(gridsizer, 0, wxEXPAND);

    // Delete device button, and button for applying changes to device details
    buttonsSizer->AddStretchSpacer();
    buttonsSizer->Add(new wxButton(this, DEVICEDELETE_BUTTON_ID, _("Delete device")), 0, (wxALL & ~wxLEFT) | wxEXPAND, 10);
    updateBtn = new wxButton(this, DEVICES_APPLY_BUTTON_ID, _("Apply changes"));
    updateBtn->Disable();
    buttonsSizer->Add(updateBtn, 0, (wxALL & ~wxLEFT) | wxEXPAND, 10);
    mainSizer->Add(buttonsSizer, 0, wxEXPAND);
  }
  else
  {
    mainSizer->Add(new wxStaticText(this, wxID_ANY, _("No device selected")));
  }

  SetSizerAndFit(mainSizer);
}

void DeviceDetailsPanel::OnApply(wxCommandEvent& event)
{
  if (!c || !selectedDev || !selectedDev->Get()) return;

  devlink d = selectedDev->Get();
  bool changedSomething = false;

  if (wxString(c->nmz()->getnamestring(d->id).c_str(), wxConvUTF8) != devicenameCtrl->GetValue())
  {
    // Update device name
    if (devicenameCtrl->CheckValid(c, d->id, true))
    {
      d->id = c->nmz()->lookup(string(devicenameCtrl->GetValue().mb_str()));
      changedSomething = true;
    }
  }

```

```
  if (d->kind == aclock)
  {
    // Update half-time-period
    if (selectedDev->Get()->frequency != spinCtrl->GetValue())
    {
      selectedDev->Get()->frequency = spinCtrl->GetValue();
      changedSomething = true;
    }
  }
  else if (d->kind == andgate || d->kind == nandgate || d->kind ==
    orgate || d->kind == norgate)
  {
    if (GetLinkedListLength(selectedDev->Get()->ilist) != spinCtrl->
    GetValue())
    {
      // Change number of gate inputs
      c->dmz()->SetGateInputCount(selectedDev->Get(), spinCtrl->
    GetValue());
      changedSomething = true;
    }
    if (selectedDev->Get()->kind != gateTypeDropdown->GetDevicekind()
    )
    {
      // Change device type between and/nand/or/nor
      selectedDev->Get()->kind = gateTypeDropdown->GetDevicekind();
      devicekindStaticText->SetLabel(gateTypeDropdown->GetValue());
      changedSomething = true;
    }
  }

  if (changedSomething)
  {
    // Notify other bits of the GUI of any changes
    c->circuitChanged.Trigger();
    c->monitorsChanged.Trigger();
    UpdateApplyButtonState();
  }
}

void DeviceDetailsPanel::UpdateApplyButtonState()
{
  // Update "apply changes" button state - only enable if field
    values are different to current device properties
  updateBtn->Disable();
  if (c && selectedDev && selectedDev->Get())
  {
    devlink d = selectedDev->Get();

    if (wxString(c->nmz()->getnamestring(d->id).c_str(), wxConvUTF8)
    != devicenameCtrl->GetValue())
      updateBtn->Enable();

    if (d->kind == aclock)
    {
      if (d->frequency != spinCtrl->GetValue())
        updateBtn->Enable();
    }
    else if (d->kind == andgate || d->kind == nandgate || d->kind ==
    orgate || d->kind == norgate)
    {
      if (GetLinkedListLength(d->ilist) != spinCtrl->GetValue())
        updateBtn->Enable();
      if (d->kind != gateTypeDropdown->GetDevicekind())
        updateBtn->Enable();
    }
  }
}

void DeviceDetailsPanel::OnInputChanged(wxCommandEvent& event)
{
  UpdateApplyButtonState();
}

BEGIN_EVENT_TABLE(DeviceDetailsPanel, wxPanel)
  EVT_TEXT(wxID_ANY, DeviceDetailsPanel::OnInputChanged)
  EVT_BUTTON(DEVICES_APPLY_BUTTON_ID, DeviceDetailsPanel::OnApply)
END_EVENT_TABLE()
```