Appendix B Framework Class Declarations

This appendix contains the framework's C++ class declarations. Anyone attempting to write a simulator should look them over.

BasicCPU Class Declaration

```
///////////////////////////////////////////////////////////////////////////////
// $Id: BasicCPU.hxx,v 1.1 1996/08/02 14:49:30 bwmott Exp $
///////////////////////////////////////////////////////////////////////////////
//
// BasicCPU.hxx
//
//   This is the abstract base class for all microprocessors
//
//
// BSVC "A Microprocessor Simulation Framework"
// Copyright (c) 1993
// By: Bradford W. Mott
// June 27,1993
//
///////////////////////////////////////////////////////////////////////////////
// $Log: BasicCPU.hxx,v $
// Revision 1.1  1996/08/02 14:49:30  bwmott
// Initial revision
//
///////////////////////////////////////////////////////////////////////////////

#ifndef BASICCPU_HXX
#define BASICCPU_HXX

#include <string>
#include <vector>

#include "AddressSpace.hxx"

class BasicCPU;
class BasicDevice;
class RegisterInformationList;
class StatisticalInformationList;
class AddressSpace;
```

```
#include "Event.hxx"

class BasicCPU {
  public:
    // Constructor
    BasicCPU(const char* name, int granularity,
        vector<AddressSpace*>& addressSpaces,
        const char* traceRecordFormat,
        const char* defaultTraceRecordEntries);

    // Destructor
    virtual ~BasicCPU();

    // Answer name of the microprocessor
    const char* Name() const { return myName; }

    // Answer the granularity of the microprocessor
    unsigned int Granularity() const { return myGranularity; }

    // Answer a reference to my event handler
    EventHandler& eventHandler() { return myEventHandler; };

    // Answer the number of address spaces used by the processor
    unsigned int NumberOfAddressSpaces() const
    { return myAddressSpaces.size(); }

    // Answer the execution trace record format
    const char* ExecutionTraceRecord() const
    { return myExecutionTraceRecord; }

    // Answer the default execution trace entries
    const char* DefaultExecutionTraceEntries() const
    { return myDefaultExecutionTraceEntries; }

    // Answer the indexed address space object
    AddressSpace& addressSpace(unsigned int addressSpace)
    { return *myAddressSpaces[addressSpace]; }

    // Execute next instruction. Answers pointer to an error message or null
    virtual const char* ExecuteInstruction(string& traceRecord, bool trace) = 0;

    // Handle an interrupt request from a device
    virtual void InterruptRequest(BasicDevice* device, int level) = 0;

    // Perform a system reset
    virtual void Reset() = 0;

    // Answers value of the program counter register
    virtual unsigned long ValueOfProgramCounter() = 0;

    // Sets named register to the given hexidecimal value
    virtual void SetRegister(const string& name, const string& hexValue) = 0;

    // Clear the CPU's Statistics
    virtual void ClearStatistics() = 0;

    // Append all of the CPU's registers to the RegisterInformationList object
    virtual void BuildRegisterInformationList(RegisterInformationList&) = 0;

    // Append all of the CPU's stats to the StatisticalInformationList object
    virtual void BuildStatisticalInformationList(StatisticalInformationList&)=0;

  protected:
    // Pointer to array of address space objects
    vector<AddressSpace*>& myAddressSpaces;

    // My event handler
    EventHandler myEventHandler;

  private:
    // My name
    const char* myName;

    // My granularity in bytes
    const unsigned int myGranularity;

    // Trace record format used by the ExecuteInstruction member function
    const char* myExecutionTraceRecord;

    // Default fields of the trace record that should be displayed by UI
    const char* myDefaultExecutionTraceEntries;
};
#endif
```

BasicDevice Class Declaration

```
///////////////////////////////////////////////////////////////////////////////
// $Id: BasicDevice.hxx,v 1.1 1996/08/02 14:50:07 bwmott Exp $
///////////////////////////////////////////////////////////////////////////////
//
// BasicDevice.hxx
//
//   This should be the base class for all devices
//
//
// BSVC "A Microprocessor Simulation Framework"
// Copyright (c) 1993
// By: Bradford W. Mott
// July 26,1993
//
///////////////////////////////////////////////////////////////////////////////
// $Log: BasicDevice.hxx,v $
// Revision 1.1  1996/08/02 14:50:07  bwmott
// Initial revision
//
///////////////////////////////////////////////////////////////////////////////

#ifndef BASICDEVICE_HXX
#define BASICDEVICE_HXX

#include <string>

class BasicDevice;
class BasicCPU;

#include "Event.hxx"

#define AUTOVECTOR_INTERRUPT  -1
#define SPURIOUS_INTERRUPT    -2

class BasicDevice : public EventBase {
  public:
    // Constructor
    BasicDevice(const char* name, const string& arguments, BasicCPU& cpu);

    // Destructor
    virtual ~BasicDevice();

    // Change my startup error message
    void ErrorMessage(const string& message)
    { myErrorMessage = message; }

    // Answer my startup error message
    string ErrorMessage() const
    { return(myErrorMessage); }

    // Answer my name
    const char* Name() const { return(myName); }

    // Answer the CPU I belong too
    BasicCPU& CPU() const { return(myCPU); }

    // Answer my initialization arguments
    string Arguments() const { return (myArguments); }

    // Answers true iff the address maps into the device
    virtual bool CheckMapped(unsigned long) const = 0;

    // Answers the lowest address used by the device
    virtual unsigned long LowestAddress() const = 0;

    // Answers the highest address used by the device
    virtual unsigned long HighestAddress() const = 0;

    // Get a byte from the device
    virtual unsigned char Peek(unsigned long address) = 0;

    // Put a byte into the device
    virtual void Poke(unsigned long address, unsigned char c) = 0;

    // Reset the device
    virtual void Reset();

    // This routine sends an interrupt request (IRQ) to the CPU
    virtual void InterruptRequest(int level);
```

```
    // This routine is called by the CPU when it processes an interrupt
    virtual long InterruptAcknowledge(int level);

  protected:
    // Reference to the CPU I belong too
    BasicCPU& myCPU;

    // My name (i.e. RAM, ROM, etc.)
    const char* myName;

    // Arguments passed to constructor
    string myArguments;

    // Error that occured during construction
    string myErrorMessage;

    // Interrupt pending flag
    bool myInterruptPending;
};
#endif
```

BasicDeviceRegistry Class Declaration

```
///////////////////////////////////////////////////////////////////////////////
// $Id: BasicDeviceRegistry.hxx,v 1.1 1996/08/02 14:50:41 bwmott Exp $
/////////////////////////////////////////////////////////////////////////////// //
// BasicDeviceRegistry.hxx
//
//   This abstract base class is used to derive a class that maintains a list
// of all the device in the simulator and allows them to be created.
//
//
// BSVC "A Microprocessor Simulation Framework"
// Copyright (c) 1993
// By: Bradford W. Mott
// October 30,1993
//
///////////////////////////////////////////////////////////////////////////////
// $Log: BasicDeviceRegistry.hxx,v $
// Revision 1.1  1996/08/02 14:50:41  bwmott
// Initial revision
//
///////////////////////////////////////////////////////////////////////////////

#ifndef BASICDEVICEREGISTRY_HXX
#define BASICDEVICEREGISTRY_HXX

#include <string>

class BasicCPU;
class BasicDevice;

// Device information structure
struct DeviceInformation {
  const char* name;           // The name of the device ("RAM","m6850",etc)
  const char* description;    // A short description of the device
  const char* script;         // UI script to get the device attachment args
};

class BasicDeviceRegistry {
  public:
    // Constructor
    BasicDeviceRegistry(const DeviceInformation* devices, unsigned int number)
        : myDevices(devices),
          myNumberOfDevices(number)
    {}

    // Destructor
    virtual ~BasicDeviceRegistry()
    {}

    // Answers the number of devices
    unsigned int NumberOfDevices() { return myNumberOfDevices; }

    // Get device information for the given index. Answers true iff successful
    bool Information(unsigned int index, DeviceInformation& information);

    // Create a device with the given name. Answers true iff successful
    virtual bool Create(const string& name, const string& args,
        BasicCPU& cpu, BasicDevice*& device) = 0;

  private:
    // Array of devices in the simulator
    const DeviceInformation* myDevices;

    // Number of devices in the simulator
    const unsigned int myNumberOfDevices;
};
#endif
```

AddressSpace Class Declaration

```
///////////////////////////////////////////////////////////////////////////////
// $Id: AddressSpace.hxx,v 1.1 1996/08/02 14:48:41 bwmott Exp $
///////////////////////////////////////////////////////////////////////////////
//
// AddressSpace.hxx
//
//   This class maintains a list of devices and provides methods to
// peek and poke into them.
//
//
// BSVC "A Microprocessor Simulation Framework"
// Copyright (c) 1993
// By: Bradford W. Mott
// June 27,1993
//
///////////////////////////////////////////////////////////////////////////////
// $Log: AddressSpace.hxx,v $
// Revision 1.1  1996/08/02 14:48:41  bwmott
// Initial revision
//
///////////////////////////////////////////////////////////////////////////////

#ifndef ADDRESSSPACE_HXX
#define ADDRESSSPACE_HXX

#include <list>
#include <string>

class AddressSpace;
class BasicDevice;

#include "BasicDevice.hxx"

///////////////////////////////////////////////////////////////////////////////
// AddressSpace class declaration
///////////////////////////////////////////////////////////////////////////////
class AddressSpace {
  public:
    // Used to retrieve information about attached devices
    struct DeviceInformation {
      string name;
      string arguments;
      unsigned int index;
    };

  public:
    // Constructor
    AddressSpace(unsigned long maximumAddress);

    // Destructor
    virtual ~AddressSpace();

    // Answer the maximum address of the address space
    unsigned long MaximumAddress() const { return(myMaximumAddress); }

    // Attach the given device. Answers true iff successful
    bool AttachDevice(BasicDevice*);

    // Detach the indexed device and destroy it. Answers true iff successful
    bool DetachDevice(unsigned int index);

    // Reset all the attached devices
    void Reset();

    // Answers the number of attached devices
    unsigned int NumberOfAttachedDevices() const;

    // Get information about the indexed device. Answer true iff successful
    bool GetDeviceInformation(unsigned int index,
        AddressSpace::DeviceInformation& info) const;

    // Peek the given location.  Answers true iff successful
    virtual bool Peek(unsigned long addr, unsigned char &c);

    // Poke the given location.  Answers true iff successful
    virtual bool Poke(unsigned long addr, unsigned char c);

  private:
    // List of attached devices
    list<BasicDevice*> myDevices;
```

```
    // Maximum address for this address space (In CPU words not bytes!!)
    const unsigned long myMaximumAddress;
};
#endif
```

BasicLoader Class Declaration

```
///////////////////////////////////////////////////////////////////////////////
// $Id: BasicLoader.hxx,v 1.1 1996/08/02 14:51:02 bwmott Exp $
///////////////////////////////////////////////////////////////////////////////
// BasicLoader.hxx
//
//   This abstract base class provides methods to load object files into the
// the simulator.
//
//
// BSVC "A Microprocessor Simulation Framework"
// Copyright (c) 1993
// By: Bradford W. Mott
// November 5,1993
//
///////////////////////////////////////////////////////////////////////////////
// $Log: BasicLoader.hxx,v $
// Revision 1.1  1996/08/02 14:51:02  bwmott
// Initial revision
//
///////////////////////////////////////////////////////////////////////////////

#ifndef BASICLOADER_HXX
#define BASICLOADER_HXX

#include <string>

class BasicCPU;

///////////////////////////////////////////////////////////////////////////////
// BasicLoader class declaration
///////////////////////////////////////////////////////////////////////////////
class BasicLoader {
  public:
    BasicLoader(BasicCPU& c)
        : myCPU(c)
    {}

    virtual ~BasicLoader()
    {}

    // Answer my CPU
    BasicCPU& CPU() { return myCPU; }

    // Loads the named file and answers an error message or the empty string
    virtual string Load(const char *filename, int addressSpace) = 0;

  protected:
    BasicCPU& myCPU;
};
#endif
```

EventHandler Class Declaration

```
///////////////////////////////////////////////////////////////////////////////
// $Id: Event.hxx,v 1.1 1996/08/02 14:51:50 bwmott Exp $
///////////////////////////////////////////////////////////////////////////////
// Event.hxx
//
//   This class maintains a queue of events requested by EventBase derived
// objects.
//
//
// BSVC "A Microprocessor Simulation Framework"
// Copyright (c) 1993
// By: Bradford W. Mott
// August 11,1993
//
///////////////////////////////////////////////////////////////////////////////
// $Log: Event.hxx,v $
// Revision 1.1  1996/08/02 14:51:50  bwmott
// Initial revision
//
///////////////////////////////////////////////////////////////////////////////

#ifndef EVENT_HXX
#define EVENT_HXX

#include <string>

class EventHandler;

///////////////////////////////////////////////////////////////////////////////
// Should be the base class for any class that is going to register
// events with the event handler
///////////////////////////////////////////////////////////////////////////////
class EventBase {
  public:
    // Constructor
    EventBase(EventHandler& handler)
        : myEventHandler(handler)
    {}

    // Destructor
    virtual ~EventBase();

    // Called when a registered event is dispatched
    virtual void EventCallback(long data, void* pointer) = 0;

  private:
    EventHandler& myEventHandler;
};


class EventHandler {
  public:
    // Constructor
    EventHandler();

    // Check for any expired events
    void Check();

    // Add an event to the event list
    void Add(EventBase* object, long data, void* pointer, long time);

    // Remove events for the given object
    void Remove(EventBase* object);

  private:
    // The event class
    class Event {
      public:
        Event(EventBase* object, long data, void* pointer, unsigned long t)
            : total_time(t), next(0),
              myObject(object), myPointer(pointer), myData(data)
        {};

        // Dispatch the event by calling the object's callback routine
        void Dispatch() { myObject->EventCallback(myData, myPointer); }

        // Return the owning object
        EventBase* Owner() { return myObject; }

        // Total amount of time to elapse before the event
```

```
        const long total_time;

        // Time left before the event occurs
        long delta_time;

        // Pointer to the next event
        Event *next;

      private:
        // The object that owns this event
        EventBase* myObject;

        // Data to be passed to the callback method
        void* myPointer;
        long  myData;
    };

    // Linked list of events
    Event* myEvents;

    // Number of calls since last time update
    long myIterations;

    // Last usec_per_check update time
    long myOldTime;

    // Average micro-seconds per call to Check
    long myMicrosecondsPerCheck;
};
#endif
```

RegisterInformationList Class Declaration

```
///////////////////////////////////////////////////////////////////////////////
// $Id: RegInfo.hxx,v 1.1 1996/08/02 14:52:55 bwmott Exp $
/////////////////////////////////////////////////////////////////////////////// //
// RegInfo.hxx
//
//   This class is used by BasicCPU (and derived classes) to manage a list of
// of register structures.
//
//
// BSVC "A Microprocessor Simulation Framework"
// Copyright (c) 1993
// By: Bradford W. Mott
// October 25,1993
//
///////////////////////////////////////////////////////////////////////////////
// $Log: RegInfo.hxx,v $
// Revision 1.1  1996/08/02 14:52:55  bwmott
// Initial revision
//
///////////////////////////////////////////////////////////////////////////////

#ifndef REGINFO_HXX
#define REGINFO_HXX

#include <string>
#include <list>

class BasicCPU;

///////////////////////////////////////////////////////////////////////////////
// RegisterInformation class declaration
///////////////////////////////////////////////////////////////////////////////
class RegisterInformation {
  public:
    // Constructor
    RegisterInformation(const string& name, const string& hexValue,
        const string& description)
        : myName(name),
          myHexValue(hexValue),
          myDescription(description)
    {}

    // Default Construtor
    RegisterInformation()
    {}

    // Destructor
    ~RegisterInformation()
    {}

    // Set the name, hex value, and the description fields
    void Set(const string& name, const string& hexValue,
        const string& description)
    { myName = name; myHexValue = hexValue; myDescription = description; }

    string Name() const { return myName; }
    string HexValue() const { return myHexValue; }
    string Description() const { return myDescription; }

  private:
    string myName;           // The name given to the register ("D0", "PC", etc)
    string myHexValue;       // The value of the register in hexidecimal
    string myDescription;    // A short description of the register
};


///////////////////////////////////////////////////////////////////////////////
// RegisterInformationList class declaration
///////////////////////////////////////////////////////////////////////////////
class RegisterInformationList {
  public:
    // Constructor
    RegisterInformationList(BasicCPU& cpu);

    // Destructor
    ~RegisterInformationList();

    // Append an element to the end of the list
    void Append(const string& name, const string& hexValue, const string& desc);
```

```
    // Return the number of elements in the list
    unsigned int NumberOfElements() const { return myList.size(); }

    // Get the element with the given index. Answer true iff successful
    bool Element(unsigned int index, RegisterInformation& info);

  private:
    list<RegisterInformation*> myList;
};
#endif
```

StatisticalInformationList Class Declaration

```
///////////////////////////////////////////////////////////////////////////////
// $Id: StatInfo.hxx,v 1.1 1996/08/02 14:53:12 bwmott Exp $
/////////////////////////////////////////////////////////////////////////////// //
// StatInfo.hxx
//
//   This class is used by BasicCPU (and derived classes) to manage a list of
// of statistics objects.
//
//
// BSVC "A Microprocessor Simulation Framework"
// Copyright (c) 1993
// By: Bradford W. Mott
// December 5,1993
//
///////////////////////////////////////////////////////////////////////////////
// $Log: StatInfo.hxx,v $
// Revision 1.1  1996/08/02 14:53:12  bwmott
// Initial revision
//
///////////////////////////////////////////////////////////////////////////////

#ifndef STATINFO_HXX
#define STATINFO_HXX

#include <string>
#include <list>

class BasicCPU;

///////////////////////////////////////////////////////////////////////////////
// The Statistic Information Class
///////////////////////////////////////////////////////////////////////////////
class StatisticInformation {
  public:
    // Constructor
    StatisticInformation(const string& statistic)
        : myStatistic(statistic)
    {}

    // Default constructor
    StatisticInformation()
    {}

    // Destructor
    ~StatisticInformation()
    {}

    // Set the statistic fields
    void Set(const string& statistic)
    { myStatistic = statistic; }

    // Answer my statistic
    string Statistic() const { return myStatistic; }

  private:
    string myStatistic;
};


///////////////////////////////////////////////////////////////////////////////
// The Statistical Information List Class
///////////////////////////////////////////////////////////////////////////////
class StatisticalInformationList {
  public:
    // Constructor
    StatisticalInformationList(BasicCPU& cpu);

    // Destructor
    ~StatisticalInformationList();

    // Append an element to the end of the list
    void Append(const string& statistic);

    // Answer the number of elements in the list
    unsigned int NumberOfElements() const { return myList.size(); }

    // Get the element with the given index. Answer true iff successful
    bool Element(unsigned int index, StatisticInformation& info);

  private:
```

```
    list<StatisticInformation*> myList;
};
#endif
```

Interface Class Declaration

```
///////////////////////////////////////////////////////////////////////////////
// $Id: Interface.hxx,v 1.1 1996/08/02 14:52:26 bwmott Exp $
///////////////////////////////////////////////////////////////////////////////
// Interface.hxx
//
// This is the user interface command class.  It handles all of the
// command's issue by the user interface.
//
//
// BSVC "A Microprocessor Simulation Framework"
// Copyright (c) 1993
// By: Bradford W. Mott
// October 21,1993
//
///////////////////////////////////////////////////////////////////////////////
// $Log: Interface.hxx,v $
// Revision 1.1  1996/08/02 14:52:26  bwmott
// Initial revision
//
///////////////////////////////////////////////////////////////////////////////

#ifndef INTERFACE_HXX
#define INTERFACE_HXX

#include <iostream.h>

class BasicCPU;
class BasicDeviceRegistry;
class BasicLoader;
class BreakpointList;

class Interface {
  public:
    // Constructor
    Interface(BasicCPU& cpu, BasicDeviceRegistry& registry,
        BasicLoader& loader);

    // Command loop
    void CommandLoop();

  private:
    // Structure for the interface's command table
    struct CommandTable {
      const char* name;
      void (Interface::*mfp)(char*);
    };

  private:
    // Indicates the number of commands in the command table
    const unsigned int myNumberOfCommands;

    // Reference to the CPU I'm managing
    BasicCPU& myCPU;

    // Reference to the device registry
    BasicDeviceRegistry& myDeviceRegistry;

    // Reference to the loader
    BasicLoader& myLoader;

    // Reference to the input stream used to get information from the UI
    istream& myInputStream;

    // Reference to the output stream used to send information to the UI
    ostream& myOutputStream;

    // Breakpoint list to manage the breakpoints
    BreakpointList& myBreakpointList;

    // Execute the given command
    void ExecuteCommand(char* command);

    // Table of commands
    static CommandTable ourCommandTable[];

    // Member funtion for each of the commands
    void AddBreakpoint(char* args);
    void AttachDevice(char* args);
    void ClearStatistics(char* args);
    void DeleteBreakpoint(char* args);
```

```
    void DetachDevice(char* args);
    void FillMemoryBlock(char* args);
    void ListAttachedDevices(char* args);
    void ListBreakpoints(char* args);
    void ListDevices(char* args);
    void ListDeviceScript(char* args);
    void ListExecutionTraceRecord(char* args);
    void ListDefaultExecutionTraceEntries(char* args);
    void ListGranularity(char* args);
    void ListMemory(char* args);
    void ListMaximumAddress(char* args);
    void ListNumberOfAddressSpaces(char* args);
    void ListRegisters(char* args);
    void ListRegisterValue(char* args);
    void ListRegisterDescription(char* args);
    void ListStatistics(char* args);
    void LoadProgram(char* args);
    void ProgramCounterValue(char* args);
    void Reset(char* args);
    void Run(char* args);
    void SetRegister(char* args);
    void SetMemory(char* args);
    void Step(char* args);
};
#endif
```