THE 6811 WOOKIE PROGRAMMING GUIDE

Submitted to: Dr. Barnicki

Submitted by: CS-401 Group 2

Kalle Anderson

Jason Buttron

Paul Clarke

Matt Enwald

Date submitted: Feb. 10, 1998

TABLE OF CONTENTS

6811 Wookie Simulator Overview

The 6811 Wookie program consists of two main components: the M68HC11 core emulator (referred in this document as the HC11) and the graphical user interface (GUI). Each component was created and built separately. The HC11 component was built into a library which the GUI component links with when it is built. Because the two components are not built into one component, there are methods for the GUI component to interface with the HC11 component. This document will help in describing how each component was written and the interface of the two components.

From now on in this document, the term HC11 will refer to the M68HC11 core emulator component that is built into a separate library. The term GUI will refer to the 6811 Wookie graphical user interface that was built under Microsoft Visual C++® with MFC support.

The HC11 Library

The entire operation of the HC11 emulator is handled inside the HC11 library. The following section describes how the library was written and how it can be used.

Class Structure

The HC11 contains many classes. The most important class is the HC11 class. This class encapsulates the entire HC11 emulator.

Creating a HC11 object

An HC11 object can be created by declaring an instance of the object. The constructor will take care of creating any internal objects. Because this HC11 object is capable of being configured in many ways, an easier way to use the HC11 object is to use the HC11Sim object. This object is designed to encapsulate most of the HC11 configuration. The HC11Sim object is covered shortly in the *HC11Sim Interface* section of this manual. Below is a simple example of declaring an instance of the HC11 object.

```
// HC11 Code Example
// Creating an HC11 object
#include "hc11.h"
…
HC11 hc11;
…
```

Configuring the HC11

There are basically two different methods of configuring the HC11. One way is to create an HC11 class and doing a number of configurations manually by calling functions and setting variables. The other way is to use the HC11Sim class to handle the entire HC11 configuration. The HC11Sim class only has a few member functions that deal with the details of the HC11 configuration.

Reset

The reset function of the HC11 takes the mode of operation as an input argument. The modes of operation can be either one of SINGLE_CHIP, EXPANDED, BOOTSTRAP, and SPECIAL_TEST modes. The reset function will initialize all of the registers of the CPU.

```
// HC11 Code Example
// Reset
#include "hc11.h"
…
HC11 hc11;                  // create HC11 object
Mode mode = BOOTSTRAP;      // Define the mode
Hc11.Reset(mode);           // Reset the HC11
…
```

Memory

Memory is represented by memory objects that are derived from the MemoryObject class. Two derived classes that are already created and used are the RAM and ROM objects, (they are covered shortly). A memory object represents a range of memory addresses in the memory map. When an instance of the HC11 object is created, a memory object is created for the 256 bytes of on-chip RAM. Also, a memory object is created for the 64 byte area of memory for the registers as well as the region where the vectors are located. All the other memory objects need to be added manually.

Adding a Memory Object

To add a memory object, an instance of a derived class from the MemoryObject class must be instantiated. This derived class must overload the Read() and Write() virtual functions. Plus, it needs to define the base address and the size somehow. Once a memory object is created with a memory range, that object can be added to the HC11 memory map object via the AddMemoryObject() function of the memory map member. If the range specified by the new memory object overlaps with a pre-existing memory object, the pre-existing memory object will take priority for memory access.

The RAM and ROM Objects

If the memory object is not an external device, it is suggested to use the memory map objects include in the HC11 library. These objects include RAM and ROM. These objects already contain means of setting the base address and the size of the memory object. Also, they contain Read() and Write() functions. Any memory object being added to the HC11 needs to be dynamically created using the new operator. This is necessary because the HC11 memory map will delete all memory objects in the destructor.

```
// HC11 Code Example
// Adding a memory object – external RAM
#include "memory.h"
…
int base_address = 0x8000;
int size = 0x8000;                         // Make a 32k RAM block
RAM* pRAM = new RAM(base_address, size);  // Dynamically create a memory
object
Hc11.memory.AddMemoryObject(pRAM);         // Add to HC11 memory map
…
```

Loading S19 Files to Memory

A program cannot be executed if the program is not first loaded into memory. The global LoadS19File function makes loading in programs easy. The function takes only one input argument, the name of the S19 file to load.

```
// HC11 Code Example
// Loading a S19 File
#include “hc11.h”
…
HC11 hc11;
LoadS19File(“car.s19”, &hc11);    // Takes the address of the hc11
…
```

Setting the Execution Starting Address

The last step before running a program in the HC11 is to set the starting address. Inside the HC11 class, there is no formal way to tell the HC11 where program execution will begin. The way to set the start address is to write the starting address of the program block into the RAM location where the reset vector is located. The reset vector is either at 0xFFFE or 0xBFFE depending on the operating mode.

The HC11Sim class does have a formal way of setting the start address and takes care of writing to memory.

HC11Sim Interfacing

Instead of manually setting the operating mode and adding external memory, a separate class will handle those details with minimal interaction with the user or programmer. The HC11Sim class creates its own HC11 object and configures it. The following section describes how to configure and use the HC11 library with the HC11Sim class.

In order to use the HC11 correctly, the functions described below MUST be used in the specified order.

Creating an instance of the HC11Sim Class

The HC11Sim class encapsulates the whole emulator (it physically contains an instance of the HC11 class). This class also contains the initialization functions needed to setup the emulator for simulation. As a result, the first step in using the emulator is to create an instance of the HC11Sim class.

```
// HC11 Code Example
// HC11Sim creation
#include “hc11sim.h”
…
HC11Sim sim;              //create an instance of the HC11Sim Class
…
```

Set the Configuration

There are a few hc11 configurations already defined in the HC11Sim class. These configurations are Briefcase, RugWarriorBootstrap, RugWarriorExpanded. To set a configuration, use the Config() function with the HC11 mode as the argument.

```
// HC11 Code Example
// HC11Sim Configuration – Briefcase Mode
#include “hc11sim.h”
…
sim.Config(BriefCase);            //Configure the HC11 Object
…
```

Load the S19 File into Memory

Now, the emulator is set up, and can run a program. To run a program, the S19 file must be loaded into memory. This is easily done using the LoadS19File() function found in the HC11Sim class. The argument to this function is just the name of the S19 file to be loaded.

```
// HC11 Code Example
// HC11Sim Loading a S19 File
#include “hc11sim.h”
…
sim.LoadS19File(TestPrg.S19);    //Loads the S19 file to memory
…
```

Set the Program Starting Address

The starting address must be set next. This is an implicit step that is not automated when the S19 file is loaded because the S19 file just loads up memory; there is no sure way of figuring out the intended starting address of a program. To set the starting address, the SetStartAddress() function found in the HC11Sim class is used. This function just places the starting address in the reset vector.

```
// HC11 Code Example
// HC11Sim Set Starting Address
#include “hc11sim.h”
…
sim.SetStartingAddress(0xC000);   //sets the starting address
…
```

Reset the HC11

Lastly the HC11 class must be reset. To do this, just call the Reset() function of the emulator; the instance of the HC11 class is public. The Reset() function takes care of many things, but the most important duty is to place the reset vector location in the PC. Now, when the Step() function is called for the first time, the emulator can start up.

```
// HC11 Code Example
// HC11Sim Reset
#include “hc11sim.h”
…
sim.hc11.Reset();         //Reset the HC11 object
…
```

Program Execution

Once the HC11 object has been instantiated, configured, and loaded with a program, the program can now be executed. Executing a program is very simple. Calling the Step() function of the HC11 will run the current instruction in memory or resolve the next pending interrupt. The HC11 will handle all of the clocking and change the program counter. No other functions need to be called to execute a program. Below is an example of using the Step() function.

```
// HC11 Code Example
// HC11 Step
#include “hc11.h”
…
for (int I=0; I < 100; I++)       // Run for a specified time
    hc11.Step();                  // Run one instruction or resolve one interrupt
```

…

Port Input/Output

Ports are part of the memory block. They can be read and written to through the memory map. Then, the memory map notifies the port and its pins. The port and pins decide if they can be changed, and if so, what happens. The 6811 Wookie project contains graphical dialogs which represent ports. These dialogs do not use the memory map to obtain information from a port. This method would require some style of polling which is inefficient. Instead, special port connectivity was built into the 6811 Wookie project to allow the port itself to contact an outside object to report information. The following section explains the function of these special port operations.

Providing Input to a Pin

To provide an input to a port pin, the external modules do need to gain access to the HC11 class object. Once the HC11 object is obtained, the PinInput() function can be called to provide an input signal to a pin. As of now, the only levels that a pin can take are boolean values, low or high (0 or 1).

```
// HC11 Code Example
// Port Input
#include “hc11io.h”
…
for (int I = 0; I < 8; I++)           // Loop through all 8 pins
    hc11.regfile.PortE.PinInput(I,1); // Provide a high input
…
```

Port Output

One way of showing port output is to poll the port by constantly reading the value from memory. Because this way is inefficient, the PortConnection class was developed. The ports of the HC11 can be setup to report to a PortConnection object every time a port is updated.

Using the PortConnection Class

The PortConnection class is actually intended to be a base class for a user-defined class. The PortConnection class contains a Write() virtual function. The write function contains an argument which represents the current value of the port. Once a user defined class is derived from the PortConnection class, the Write() function can be overridden to suit the user’s needs. The write function should never be called in the external object’s code except for special cases; the port object inside the HC11 will do the function calling which will drive the output displays.

The PortConnection object is actually connected to a port by calling the port’s Attach() function. Once the object is attached, the object will start receiving function calls from the port if the port is being updated. The PortConnection can be detached by calling the port’s UnAttach() function.

If a PortConnection class is attached to a port and the write function has not been overridden, the PortConnection object has no effect.

```
// HC11 Code Example
// PortConnection Usage
#include “hc11io.h”
…
// Derive a class from PortConnection
class ShowPort : public PortConnection
{
    Write(byte_t value);      // Overriding the PortConnection::Write function
    DisplayToScreen(byte_t value);    // Displays a byte value to a screen
};
…
// Attaching to a port
ShowPort showport;           // Create an instance of the PortConnection object
hc11.regfile.PORTB.Attach(&showport); // Attach the object to the port
…
// Body of Overridden Write Function
void ShowPort::Write(byte_t val)
{
    DisplayToScreen(val);     // val holds the current value of the port
}
```

The PinConnection Class

The PinConnection class very similar to the PortConnection class. The only difference is that a PinConnection object connects to a single pin rather than a whole entire port. All of the implementation is the same as PortConnection. The HC11 allows PinConnection objects attached to pins on a port that already has a PortConnection.

Customizing Register Functions

In many cases with the M68HC11 micro-controller, writing or reading to particular registers will have effects on other systems within the processor. On the real HC11 chip, this is done by wiring the register to other systems. In the 6811 Wookie project, the systems are made to be object-oriented and oblivious to the other systems that exist around it. This makes it more difficult to allow systems to work together.

The Register Architecture Problem

More specifically, all register read and write functions were designed to run under one base class, the ByteRegister class. Inside the write function of this class, there is no way of knowing which register is being updated.

In the HC11 library, all registers are objects of the ByteRegister class. These registers reside as member variables of the RegisterFile class. With this architecture, there is no way for a register to affect another system of the HC11 when the register is being changed. More specifically, the ByteRegister has no way of gaining access to any other objects within the HC11.

The Solution

To overcome this problem, some registers have been customized. Instead of the registers being member variables of the RegisterFile class, some registers are member *classes* of the register file. When a new class is made for a particular register, the read and write functions can be overridden to perform special operations. To gain access to other areas of the HC11, access is given to the custom register classes when they are created.

In this way, the plain ByteRegister still does not have access to any other system. This preserves the object-oriented model. With the custom registers, they have member variables that are pointers to other systems of the HC11 that may need to be updated. The following code shows an example of a custom register class.

```
// HC11 Code Example
// Customizing the TCTL1 Register
// The TCTL1 Register enables/disables the output compares
// The TCTL1 Register needs to affect PORTA’s pins

class RegisterFile
{
public:

    class TCTL1_C : public ByteRegister      // Make the member class
    {
    public:
        void Write(byte_t data);             // Override the write function
        PortRegister* portA;                 // Allow access to PORTA
    };
    …
// Give TCTL1 access to PORTA
// RegisterFile constructor
RegisterFile::RegisterFile(Mode mode)
{
    TCTL1.portA = &PORTA;
    …
}
```

Not all of the registers that need to be customized are customized in the 6811 Wookie. The HC11 in 6811 Wookie does not have all of the subsystems implemented so all of the specialized register functions did not need to be implemented. If a subsystem were to be added at a later date, some registers would need to be customized in the similar way as in the previous example.

Subsystems

A subsystem in the M68HC11 is a specialized group of hardware that performs specific functions. Some examples are the timer system, the serial communications system, the serial peripheral system, and the analog to digital conversion system.

Timer Subsystem

The timer subsystem is the only subsystem built into the HC11 of the 6811 Wookie project. The following code examples show how the subsystem is created and used in the HC11 library.

```
// HC11 Code Example
// The Timer Subsystem
// Found in “timer.h”

// The Timer class
class TimerSubSystem : public HC11SubSystem
{
 protected:

    HC11* hc11;// Access to HC11

 // Clock Dividers Counters
    word_t  rti_clock_divider;
    word_t  pulse_accumulator_clock_divider;

 public:
    word_t  main_clock_divider;
    void DoOutputCompares(void);
    void DoPulseAccumulator_GatedTimeAccumulationMode(void);

TimerSubSystem(HC11 * nhc11);
    void ClockSystem(void);
};
…
// HC11 _clock function
void HC11::_clock()
{
timer_system->ClockSystem();
    timer_count++;
};
```

Notice that the timer subsystem has access to the complete HC11 class. This presents a small security risk, but was the easiest way to give the subsystem access to many different HC11 components. To make the HC11 more secure, the HC11 variable is protected within the HC11Subsystem class. This way, no other class may use the subsystem to change the HC11. The HC11 can only be changed by the subsystem member functions.

Also, notice that the subsystem has member function that perform special functions, such as the output compares and the pulse accumulator.

Adding 6811 Wookie Subsystems

Subsystems in the 6811 Wookie project are classes derived from the HC11SubSystem class. A HC11SubSystem class should never be instantiated. This class was built to be overridden by a derived class. When a subsystem is created, it is added to the HC11 architecture by adding a member variable to the HC11 class.

```
// HC11 Code Example
// HC11SubSystem class definition
// Found in “systems.h”

class HC11SubSystem
{
 protected:
    class HC11 *hc11;
 public:

    HC11SubSystem(){hc11=NULL;};
    HC11SubSystem(HC11 *nhc11);
    void ClockSystem(void);

 friend class HC11;
};

// Creating your own subsystem
class ADConverter : public HC11SubSystem
{
    …
};

// Add to HC11 class
class HC11
{
    …
    ADConverter AD; // Add as a member variable
    …
};
```

Once the subsystem is added, any special functionality must be integrated into the operation of the HC11. For instance, the timer subsystem functions from the HC11 E-clock. Inside the clock function of the HC11, the HC11 calls the timer subsystem clock function. In this way, the timer subsystem will always receive a clock signal when the HC11 does.

If the subsystem requires special functionalities of registers, then the register may need to be customized. For instance, the serial communications subsystem can generate an interrupt when a byte is received. Therefore, the register that accepts the byte must have a way of generating interrupts. This can be done by customizing the register. For more information on how to customize a register, see the section titled *Customizing Register Functions*.

THE 6811 WOOKIE GUI

The 6811 Wookie GUI was created using the Microsoft Foundation Classes (MFC) AppWzard. The project is a single-document interface with context sensitive help. This document does not contain any details on the created of the individual dialogs that accompany the main views of the 6811 Wookie. This document does contain, however, the necessary information to re-create the same layout of the 6811 Wookie and be able to run a program in the same fashion.

Main Window Layout

During the start-up initialization of the project, the option of creating a splitter window was chosen. AppWizard then created the static splitter window. The splitter window separates the main window into two views. The two view classes that represent these two views are the HC11View class and the CodeView class. The creation of the splitter window views occur in the CMainFrame::OnCreateClient function.

```
// GUI Code Example
// Adding splitter windows and views

BOOL CMainFrame::OnCreateClient( LPCREATESTRUCT /*lpcs*/,
    CCreateContext* pContext)
{
    // Creating the horizontal splitter bar to divide the main dlg with the Code View
    if (!m_wndSplitter.CreateStatic(this,2,1))
    {
            return FALSE;
    }
    m_wndSplitter.CreateView(0,0,RUNTIME_CLASS(CHc11View),CSize(380,220),pContext);
    m_wndSplitter.CreateView(1,0,RUNTIME_CLASS(CCodeView),CSize(0,0),pContext);
    SetActiveView((CView*)m_wndSplitter.GetPane(0,0));
    return TRUE;
}
```

Both views share the same document class, the CHardCoreWookieDoc class. All of the data concerning the HC11 is stored as data members in the document.

The HC11 View

The HC11 view is the upper view of the main window and contains all of the buttons and controls that operate the HC11. Included among these controls are the MCU watch button, the Register watch button, the Memory set button, the IRQ and XIRQ buttons, the Break-point text box, the Pin Scope button, the Start/Stop button, the Step button, and all of the port buttons.

The main functionality of the HC11 view is to operate the HC11. The execution of the HC11 can be controlled using the Start/Stop button or the Step button. A program can be halted on a specific line of code using a break point. All the other buttons bring up separate dialogs that perform special functions.

All of the separate dialogs, except the memory set dialog, were created as pop-up style dialogs. This style of dialog allows the dialog to be visible without disabling the main dialog that created it.

The Code View

The code view is the lower view of the main window and is based from a CEditView class. This type of view class allows text to be displayed in a view easily and is accompanied by most of text edit functions such as copy, cut, and paste.

The main functionality of the code view is to display the list file and return any information of the list file. The list file is continually searched to match up the current instruction of execution to the correct line of text. Other information such as the ORG statement and any RMB variables are extracted from the list file.

Another item of note is that the program is partially dependent on the file format of the list file. When a list file is initially loaded, the code view is searched to find the header of the list file to determine the file format. If the file format is known, the code view will use the appropriate search routine to highlight text in the view. If the file format is not known, then the highlighting functionality will be unavailable for that program.

As of date, only two list file formats are supported, the AS11M compiler list format and the GCC compiler list format. Other list formats may be supported as they are brought to attention.

Program Execution

A program is executed in two different ways on the 6811 Wookie: step execution or start/stop execution.

Step Execution

Using step execution, a program will execute one instruction at time. The line of code in the list file that is about to be executed will be highlighted in the code view. All of the watch information in the separate dialogs is updated on every step. The 6811 Wookie does not use a separate thread for step execution.

```
// GUI Code Example
// Step Execution
// This function is called when the Step button is pressed

void CHardCoreWookieDoc::OnSimmulatorStep()
{
    if (!(m_pCodeView->m_isFileOpen))
        AfxMessageBox("Must Load an S19 file first!");
    else
    {
        // Call the step function once
        // Reset the highlighted text
        hc11sim.hc11.Step();                     // Steps the HC11
        m_pCodeView->SelectLine(&hc11sim.hc11);  // Highlights the next line
        m_pHc11View->Update();                   // Updates the watches
    }
}
```

Start/Stop Execution

Using the start/stop execution, the program will run at full speed. The HC11 has no timing delays built in so the execution will be running as fast as it can go. To leave the main dialog free for mouse events, the program execution is handled in a different thread.

The thread used is a worker thread. The function that the thread runs is called TimerProc. This thread is created at the very beginning of when the 6811 Wookie program is launched. The thread is paused immediately and will remain paused until the start/stop button is pressed. Once the thread starts execution, the thread steps the HC11 at full speed until the execution is stopped by the start/stop button or a break point. When the execution is stopped, the m_isrunning variable is cleared which forces the thread to fall out of its stepping loop. When the thread stops stepping, the thread pauses itself and sends a message to the view class informing that execution has stopped. The thread will again remain paused until the start/stop button is pressed.

The thread is paused and resumed using an Event object. The thread will wait for a signal from the event object that the thread is ready to execute. When the thread is finished executing, the thread resets the event object to force the thread to continue waiting for the even object to be signalled.

```
// GUI Code Example
// Worker Thread Function TimerProc

UINT TimerProc(void * pParam)
{
    CHardCoreWookieDoc* pDoc = (CHardCoreWookieDoc*)pParam;

    // Loop forever
    for(;;)
    {
        // This is used to pause the thread
        WaitForSingleObject(pDoc->m_hThreadGo,INFINITE);

        while(pDoc->m_isrunning)
        {

            pDoc->hc11sim.hc11.Step();        // Step the HC11

            // Stop on a breakpoint
            if (pDoc->m_breakpoint == (int)pDoc->hc11sim.hc11.PC)
                break;
        }
        // Reseting the event will "pause" this thread again.
        ResetEvent(pDoc->m_hThreadGo);

        // Notify the Hc11View that the thread has stopped stepping
        // the HC11.
        ::PostMessage(pDoc->m_hwndHc11View,WM_THREAD_STOP, 0, 0);
    }
    return(0);
}
```