Embedded Systems Simulator

Simulation of Microcontroller and Peripherals using Plug-in Architecture

THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

By Nicholas Reynolds

School of Information Technology

and Electrical Engineering

University of Queensland

Submitted for the degree of Bachelor of Engineering (Honours)

in the division of Software Engineering

24th October 2007

Nicholas Reynolds

24 October 2007

Head of School,
School of Information Technology and Electrical Engineering,
The University of Queensland,
St Lucia QLD 4072

Professor Paul Bailes,

In accordance with the requirements of the Degree of Bachelor of Engineering (Honours) in the School of Information Technology and Electrical Engineering, I submit the following thesis entitled

"Embedded Systems Simulator".

This thesis was performed under the supervision of Dr Peter Sutton. I declare that the work submitted in this thesis is my own, except as acknowledged in the text and footnotes, and has not been previously submitted for a degree at the University of Queensland or any other institution.

Yours Sincerely,

Nicholas Reynolds

Abstract

Software development for embedded systems is often complicated by the lack of a standard set of debugging tools once code has been programmed to a microchip. In an effort to afford an environment in which these debugging tools can be provided, it is the aim of this thesis to enable the simulation of these systems. This reveals the hidden internal operations of the microcontroller thus allowing a these tools to be provided.

In order to fully debug these systems it must be possible not only to simulate the microprocessor but the peripherals connected to it also.

This is not the first approach to simulate embedded systems however it is the purpose of this thesis to develop a freely distributable, platform independent, extendable, easy-to-use, fast simulation environment with the ability to emulate the specific hardware required by the university.

The requirements and approach of creating such a system is explored through the analysis of current simulation techniques and existing technologies. Also included is an expected project schedule, breakdown of required tasks, and risk assessment.

Acknowledgements

First and foremost I would like to acknowledge my supervisor, Dr Peter Sutton, for his inspiration, guidance and advice throughout the duration of this thesis. I would also like to extend thanks to my friends with special thanks to:

- James McGill for sharing his expertise on Atmel microcontroller architecture
- Nathan Kachel for sharing his knowledge of the thesis experience
- Jonathan Thompson for always keeping an eye on my progress

I would also like to thank my family, specifically my mother for her love and support, and my father for his love and his determination to see me finish.

Finally, thanks go to my wonderful girlfriend for the support she offered and her unparalleled level of understanding, especially during the prolonged periods we spent apart.

Table of Contents

List of Figures

1 Thesis Overview

1.1 Introduction

Embedded systems can be found in almost every appliance in modern society, from the controller in a home microwave to the circuit in your wrist watch. These systems are becoming more common as devices are required to perform more complicated tasks. Unfortunately, the development of software for these systems can be a long, arduous task due to innate difficulties such as the lack of a comprehensive debugging environment. Often the testing of these systems is undertaken by measuring external voltage levels and pin outputs. The aim of this thesis is to create a complete simulation environment in which microcontroller code can be debugged using systems traditionally unique to software development for a personal computer. Furthermore, the environment will support the addition of peripherals using an open plug-in architecture. This will allow the creation of future components with minimal effort.

The creation of a simulation environment for embedded systems is not a revolutionary concept. There is a variety of systems already in existence which offer similar functionality to that which is being proposed, however, these do not meet the university's needs. Problems with these systems include slow simulation speeds, the inability to use peripherals, incompatibility with the multiple operating systems, costs of licensing, and lacking support for the required hardware. The primary function of the proposed simulator is to allow students to develop software for embedded systems without needing access to laboratory equipment, thus enabling students to work from any PC at any time. In addition, source code for the developed simulation environment will be available, allowing custom evolution and adaption to future needs of the university.

1.2 Project Motivation

The primary motivation for creating the new Embedded Systems Simulator is to create a tool which can accurately simulate the behaviour of the hardware used by students of the "Introduction to Computer Systems" course offered by The University of Queensland. This will allow students to develop software for embedded systems without access to laboratory equipment, thus enabling students to work from any PC at any time. In addition, the simulator should be designed to facilitate evolution and adaption to future needs.

1.3 Outcomes

The project, while providing many large improvements over previous version, is currently not in a usable state for the students of Introduction to Computer Systems. Some minor bugs remain and two Component objects are missing. Other than these two shortcomings, the system provides a fast, portable simulation and debugging environment capable of disassembling byte code, controlling the execution of the code, and viewing the entire internal state of the AT90S8515. It allows users to institutively customise, add, and connect components.

Work will continue on this project in the immediate future and all changes will be propagated back to the University of Queensland.

1.4 Thesis Outline

This document presents the background, the aims, the achievements, the architectural design, and an evaluation of the Embedded System Simulator.

Chapter 2 introduces the topic of simulation and describes some of the concepts important to understanding simulation, and some products that are currently available with similarities to the goals of the Embedded System Simulator.

Chapter 3 identifies the intended goals and features of the final product.

Chapter 4 describes the implementation and the design of the Embedded System Simulation, detailing the structure and the main components.

Chapter 5 overviews the product outcomes and identifies the strengths, weaknesses and completeness of the new design. This chapter also identifies future improvements which could be possibly be set as future thesis topics.

Finally, chapter 6 provides a summary of this thesis.

2 Literature Review

This research provides information on the selection of a simulation technique and details of current simulation software offering similar functionality. Reviewing existing simulators allows the identification of capabilities and functionalities which could be included in this thesis, as well as highlights the flaws in these simulators thus identifying areas where improvement can be made. Through the identification and analysis of this information, it is possible to draw from the strengths and learn from the weaknesses, ultimately improving the design of this thesis.

The review of previous products is limited to purely software based simulation as hardware-software hybrids are beyond the scope of this thesis.

2.1 Simulation Techniques

There is a variety of concepts for modelling simulations each with distinct advantages. The chosen type for this embedded system simulator is discrete-event, deterministic simulation. Following are explanations of this decision.

2.1.1 Discrete Event

A discrete event simulation manages events in time [1, 4]. In this model, a queue, ordered by time, is kept of each event to occur. The process of running this simulation then consists of two steps [5].

1. Advance to the simulation time (not wall clock time) of the next event
2. Execute all events scheduled for current time

The first step is achieved by advancing time to that of the event at the head of the queue (always the earliest event pending). The next step includes executing all events in the queue with the same simulation time. As a result events may be scheduled in the future. Using this design allows time between events to be ignored while maintaining the accurate simulation of devices with specific timing requirements (such as Dallas One-Wire devices which use a time sensitive asynchronous protocol [6]).

> Example: Executing an "out" instruction event for a microprocessor might enqueue two events.
>
> 1. Change the output pin of the chip

2. Schedule an event for the next instruction to be executed

These would both be scheduled at the same time which would be the current simulation time plus the time it takes for one clock cycle of the processor.

There are many techniques to improve performance within this method [3, 5, 7-9] including space or time efficient data types for the event queue, faster transformation algorithms, and multi-threading of events. These will all be considered when creating the logic for the simulator engine.

An alternative simulation technique is continuous simulation [10], in which time is continually incremented, in significantly small steps to appear continuous, after which the states of all components are updated. This approach has been shown to have a lower performance than the discrete event-based method for the simulation of microprocessors [11].

2.1.2 Determinism

In many simulation environments there is not one defined transformation from one state to another. Instead a probability distribution is used to determine the likeliness of each resulting state in a set of states. This is known as the stochastic model, the inverse of which is the deterministic model. A deterministic model describes one possible outcome per event. This is the model that will be used in the embedded systems simulator due to the native deterministic qualities of digital systems.

2.1.3 Instruction versus Hardware Simulation

There are two main methods for simulating the internal logic of components. The first is to simulate the entire hardware system, modelling the interaction between the electrical components. This approach tends to have a reduced simulation speed [12], much slower than the real-time constrains upon this thesis. The second method entails the abstraction of the internal operations, allowing much faster speeds but reducing the accuracy of which the microcontroller behaviours are simulated. This is a necessary trade-off to maintain real-time simulation and is thus the technique to be used for this thesis.

2.1.4 Languages

There are a multitude of languages which could be used to create an embedded systems simulator, some specifically languages for simulation such as GPSS, SimPy and Simula. There is a large move to produce simulation using Java [13-17] with some of the advantages including native multithreading support, the ability to integrate into a web browser and be distributable online, multiplatform compatibility, and the easy code and component reuse gained from object oriented abstraction. The main disadvantage of Java is the hypothetical decrease in simulation speed due its interpretive nature [14] however this has been combated, stating that Java should offer sufficient speeds for these purposed [16].

2.1.5 Current Simulation Packages

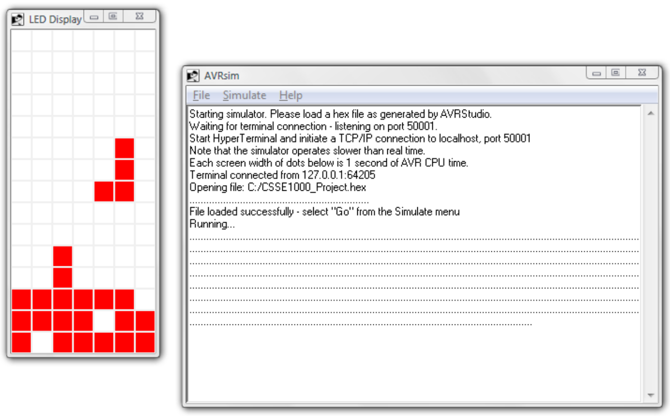
Simulation libraries such as SML and Silk have been developed for the Java framework [13, 14, 16, 18, 19]. These libraries provide methods and interfaces for process based discrete-event simulation and may prove useful during the creation of the Embedded Systems Simulator. As such they will be explored further during the design of the simulator logic.

2.2 Prior Art

As stated, there is already a large collection of embedded system simulators currently available which closely resemble the desired final product of this thesis. Following is an analysis of a selection of these programs. Packages which rely on software-hardware hybrid techniques, such as In-Circuit Emulators (ICE), are not included in this study due to their innate lack of portability.

2.2.1 Embedded System Simulator

The Embedded System Simulator (ESS), a simulator for the AT90S8515 and ATMEGA128 microprocessors, was developed by Dr Peter Sutton and extended by Mr James Kehl. The ESS is capable of accurately simulating all of the microprocessors instructions and supports the addition of peripherals to the simulation. This system lacked debugging functionality and a user interface for peripherals.



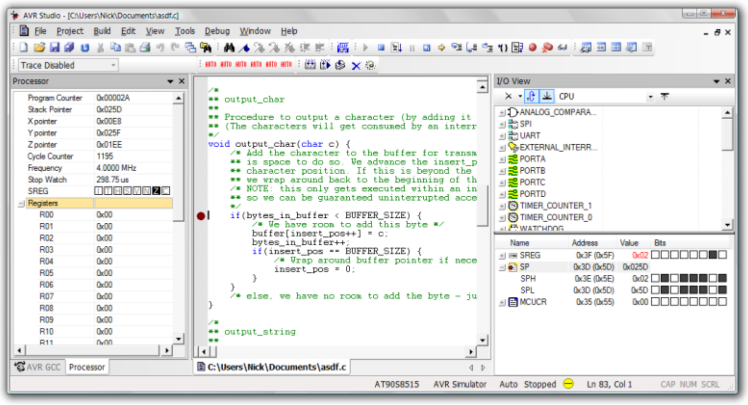
```
Starting simulator. Please load a hex file as generated by AVRStudio.
Waiting for terminal connection - listening on port 50001.
Start HyperTerminal and initiate a TCP/IP connection to localhost, port 50001
Note that the simulator operates slower than real time.
Each screen width of dots below is 1 second of AVR CPU time.
Terminal connected from 127.0.0.1:64205
Opening file: C:/CSSE1000_Project.hex
............................................................
File loaded successfully - select "Go" from the Simulate menu
Running...
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
.................................................................................................................
```

Figure 1 – The Embedded System Simulator Running a Game of Tetris

This simulator is currently used by the students of Introduction to Computer Systems. The choice to use this simulator, as opposed to others and later versions of this application, is due mainly to the ease of creating extensions by the course coordinator. Figure 1 shows the latest update to the system, built to simulate the final project for 2007.

2.2.2 AVR Studio

Atmel, the manufacturers of the microprocessor to be simulated in this thesis, provide this IDE freely on the company website. This simulator has a customizable, fully-symbolic, source-level debugger which allows breakpoints and all internal registers can be viewed graphically. It also provides limited support for plug-ins such as LCD displays. While this application fulfils many of the specified requirements it is restricted to operation on Windows operating systems. Figure 2 shows the simple yet thorough graphical representation used to display the internal state of the microcontroller. This user interface simplifies the display of the registers/data memory and is a definitely a noteworthy application for future design.



Program Counter	0x00002A
Stack Pointer	0x025D
X pointer	0x00E8
Y pointer	0x025F
Z pointer	0x01EE
Cycle Counter	1195
Frequency	4.0000 MHz
Stop Watch	298.75 us
SREG	I T H S V N Z C
Registers	
R00	0x00
R01	0x00
R02	0x00
R03	0x00
R04	0x00
R05	0x00
R06	0x00
R07	0x00
R08	0x00
R09	0x00
R10	0x00
R11	0x00

```
/*
** output_char
**
** Procedure to output a character (by adding it
** (The characters will get consumed by an interr
*/
void output_char(char c) {
    /* Add the character to the buffer for transm
    ** is space to do so. We advance the insert_p
    ** character position. If this is beyond the
    ** we wrap around back to the beginning of th
    /* NOTE: this only gets executed within an in
    ** so we can be guaranteed uninterrupted acce
    */
    if(bytes_in_buffer < BUFFER_SIZE) {
        /* We have room to add this byte */
        buffer[insert_pos++] = c;
        bytes_in_buffer++;
        if(insert_pos == BUFFER_SIZE) {
            /* Wrap around buffer pointer if nece
            insert_pos = 0;
        }
    }
    /* else, we have no room to add the byte - ju
}

/*
** output_string
**
```

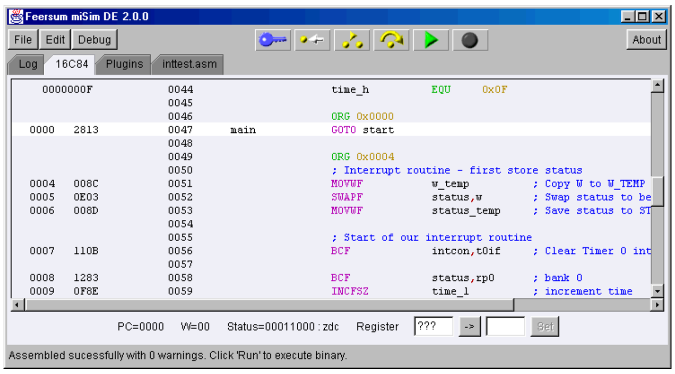
ANALOG_COMPARA...
SPI
UART
EXTERNAL_INTERR...
PORTA
PORTB
PORTC
PORTD
TIMER_COUNTER_1
TIMER_COUNTER_0
WATCHDOG

Name	Address	Value	Bits
SREG	0x3F (0x5F)	0x02	□□□□□□■□
SP	0x3D (0x5D)	0x025D	
SPH	0x3E (0x5E)	0x02	□■□■■■□■
SPL	0x3D (0x5D)	0x5D	□■□■■■□■
MCUCR	0x35 (0x55)	0x00	□□□□□□□□

Figure 2 – Screenshot of AVR Studio's Main Interface During Simulation

2.2.3 miSim DE

This Java program is designed to simulate PIC microchips and has the functionality for peripheral plug-ins. This simulator is very similar to the desired final product including its plug-in functionality, debugging capabilities, and modularity (it has the ability to run as a web applet). This simulator reached speeds nearing 100MHz on a 2GHz laptop, proving it is possible for Java to run high speed simulations.



```
  0000000F                      time_h       EQU     0x0F

                                ORG 0x0000
 0000    2813         main      GOTO start

                                ORG 0x0004
                                ; Interrupt routine - first store status
 0004    008C                   MOVWF        w_temp          ; Copy W to W_TEMP
 0005    0E03                   SWAPF        status,w        ; Swap status to be
 0006    008D                   MOVWF        status_temp     ; Save status to ST

                                ; Start of our interrupt routine
 0007    110B                   BCF          intcon,t0if     ; Clear Timer 0 int

 0008    1283                   BCF          status,rp0      ; bank 0
 0009    0F8E                   INCFSZ       time_l          ; increment time
```

PC=0000 W=00 Status=00011000 : zdc Register ??? -> Set

Assembled sucessfully with 0 warnings. Click 'Run' to execute binary.

Figure 3 – Screenshot of miSim DE's Main Display

The only features missing from this program is the ability to simulate the Atmel microchip and the lack of available source code (the source code for the newer versions of this software is unavailable). The GUI separates the main microchip debugging window from other components, shown in Figure 3, however it does not provide an intuitive way to connect multiple components together. As can be seen in Figure 4, each component in the system has an individual GUI window, cluttering the desktop during more complicated simulations.

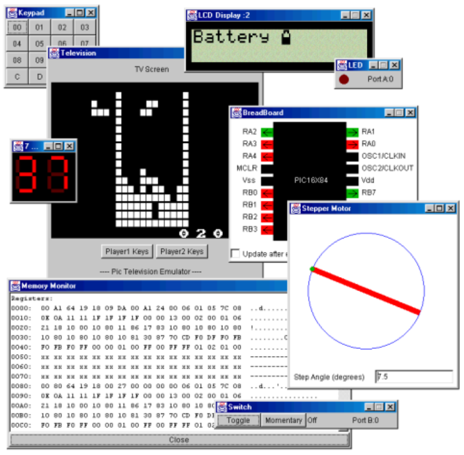


Figure 4 - A Collection of miSim DE Plugins

With the source code being freely available for this product, it will be a good point of reference for many design decisions.

2.2.4 SimulAVR

This project is under development, currently not at a stage where is can be used reliably. However, there are some noteworthy features of this program. SimulAVR [20] integrates into both WinAVR (a C compiling tool for Atmel chip code) and the GNU Debugger (GDB), utilising the features and interface. GDB is a popular debugger with a lot of powerful functions such as breakpoints, watch points, the ability to step through code and examine memory, and the facility to control execution flow. Utilising GDB decreases the learning curve significantly for users experienced with this environment. Unfortunately this program does support peripherals, has no built in graphical user interface, and is dependent on GDB.

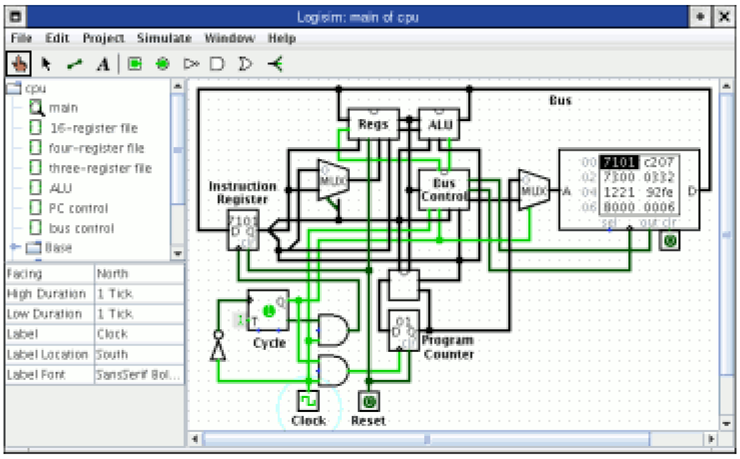
2.2.5 The Updated Embedded Systems Simulator

The Embedded Systems Simulator (ESS) was updated by thesis students, for the University of Queensland. The motivation behind this simulator was to extend the previous model, increase the number of components provided, and create a more intuitive connecting of components. This version is capable of simulating both PIC and Atmel microcontrollers as well as additional peripherals via a plug-in interface. It was written entirely in C and C++ and runs on both Solaris and Windows operating systems. The user interface interacts with the simulator logic via a TCP/IP port, this communication interface reduces the possible simulation speed significantly (however provides a useful abstraction between the GUI and simulation logic).

This simulator has many useful features which will be implemented in the new ESS. Even though it was written in a different language, having access to the source code should eliminate the some of the difficulties faced in the designing the new simulator.

2.2.6 Logisim

Logisim is a logical circuit simulation package with the capability to build circuits graphically using a drag and drop display. While this simulator differs from the others reviewed such that it simulates much more basic circuitry, this application offers an intuitive, easy to use interface. The window on the left side of the main window shown in Figure 5 provides a simple way to select components and customise the settings for each specific use.



Facing	North
High Duration	1 Tick
Low Duration	1 Tick
Label	Clock
Label Location	South
Label Font	SansSerif Bol...

Figure 5 – The Main Window for Logisim

This software package is open-source and as such the code used for the graphical user interface can be examined and techniques borrowed for the design of the new Embedded Systems Simulator.

2.3 Summary

There are many aspects of applications reviewed above which will be considered when building the new Embedded Systems Simulator. The graphical user interface used to display the internal workings of the microprocessor are well designed in AVR Studio, and the method of connecting and placing components is very intuitive in Logisim. The techniques used to provide plug-in capabilities for miSim DE will be further investigated and possibly imitated.

Some of the software packages described above are open source and as such inspection of the code is possible. It is important to note that no source code will be copied from any of these packages. The techniques observed will, at most, be used as inspiration for writing of the code for the new Embedded Systems Simulator.

3 Software Design Specification

"Simulation enables the study of, and experimentation with, the internal interactions of a complex system" – Discrete-Event System Simulation [1]

The objective of this thesis is to produce an accurate, fast, portable simulator with support for the Atmel AT90S8515 Microcontroller and a set of peripherals. It must also support a set of debugging tools for microcontroller simulation allowing full examination of internal registers as well as providing functions to create breakpoints and "step" through code execution.

3.1 General Topic Background

The main motivation for creating this simulator is to improve the software debug process for embedded systems, attempting to making it easier, faster, and in some cases cheaper. As stated earlier the main issues with developing code for these systems are:

1. Lack the ability to view the internal state
2. Have to reprogram chip each time code changes
3. Need circuitry and components ready for testing

Modern development of computer software is done in an environment where debuggers are readily available. These debuggers allow the programmer to see the internal state of the program (all variables, position of execution, etc.) and step through the code line by line. In addition they can create breakpoints in the code so the program will run normally until reaching these locations at which stage execution pauses for user input. These enable programmers to identify, analyse and correct code faults in a quick and easy manner. Programming for microcontrollers does not afford these luxuries. Code changes require reprogramming of the microchip and debugging requires the use of electrical measuring devices to monitor external behaviour.

This simulator aims to extend some of the features of the modern software debugging environment to programmers of microcontrollers by allowing virtual execution of code to run on a PC. This will allow debugging features to be available and remove the need for the time consuming reprogramming of microchips. Moreover, with the ability to add other components to the simulation (push buttons, LED, MAX232) no physical circuit will be required to begin testing code, possibly reducing cost and complexity.

Simulators offering similar features and functionality to that purposed already exist, however, none meet the specific needs of the university. The requirements are for a freely distributable, multiplatform, extendable, easy-to-use, fast simulator which can emulate the specific hardware used by the university. Having such a simulator would allow students to develop embedded systems code from the comfort of any PC, at any time, without access to laboratory equipment.

3.2 Specification

This section aims to define the specifics of a system based on these requirements, defining all the aspects of the simulator.

3.2.1 Platform Compatibility

The software must be compatible with Solaris and Windows operating systems. The chosen method will be compatible with any platform supporting the Java Runtime Environment [2].

3.2.2 Software Architecture

The program should be designed using a plug-in architecture allowing simple development of additional components. Abstraction should be used where possible to allow reuse of code particularly with respect to each component's graphical user interface (specifically requested by supervisor).

3.2.3 Required Components

- Atmel AT90S8515 Microcontroller [3]
  This must be able to load, disassemble, and execute compiled HEX files. It must also provide debugging features allowing the inspection of the internal state (registers), the addition of breakpoints, and the ability to step through code (assembly) on a line by line basis. Finally, it must support the Electronically Erasable Programmable Read-Only-Memory (EEPROM) such that this memory can be loaded before starting the simulation and will be persistent through chip resets. It is not within the scope of this thesis to be able to save this data, though such functionality may be included.
  Some features which may be left out of the design of this chip (not in the scope of this thesis)

    - Analogue Comparator
    - Serial Programmable Interface (SPI)
    - Watch Dog Timer
    - Low Power/Idle Modes
- Light Emitting Diode (LED)
  This should support different colours (minimum 2).
- Push Button/Switch
  The code for these two components will be very similar. Ideally (not in the scope of this thesis) these components should simulate contact bounce.
- Seven Segment Display
  Only one interface method is required for this component.
- MAX232 Chip
  This should simulate the RS232 communication (suitable of that usually observed using HyperTerminal) including the replication of errors which would be occur due to incorrect baud rate, etc. This will be output to a TCP/IP port for usage in programs such as HyperTerminal. A feature outside the scope of this thesis would be to include a Java virtual terminal for use with this plug-in.
- Piezoelectric Speaker
  This must output sound to the user and have the option to be muted.

3.2.4 Graphical User Interface (GUI)

The user interface will be modular allowing additional components to be displayed easily, ideally combinable into saveable sets (i.e. a set of eight push buttons and LED) for future use.

Additionally, as stated earlier, ideally there would be an abstraction between the graphical and logical sections of components, to facilitating future use of the graphical interfaces.

3.2.5 Performance

Simulation should be capable of real time speeds given that a maximum of two (2) additional basic components are included with a microcontroller clocked no faster than 4MHz. The software must have the capability to throttle simulation speed down to real time if desired.

4 Implementation

This section focuses on the actual implementation of the design specification. It will describe the overview of the entire system and discuss the specifics of each component. Additionally this will detail some of the design decisions made throughout the development of the Embedded System Simulator.

4.1 Overview of Design

The focus during the design of the system has been to create a fast simulator with a simple framework to easily create and include additional components. Carefully chosen data structures and specially crafted logic in the AT90S8515 has allowed the simulator to reach real-time speeds. An easily extendable architecture has been created using a number of design techniques with a large emphasis on programming templates, and strong use of reflection.

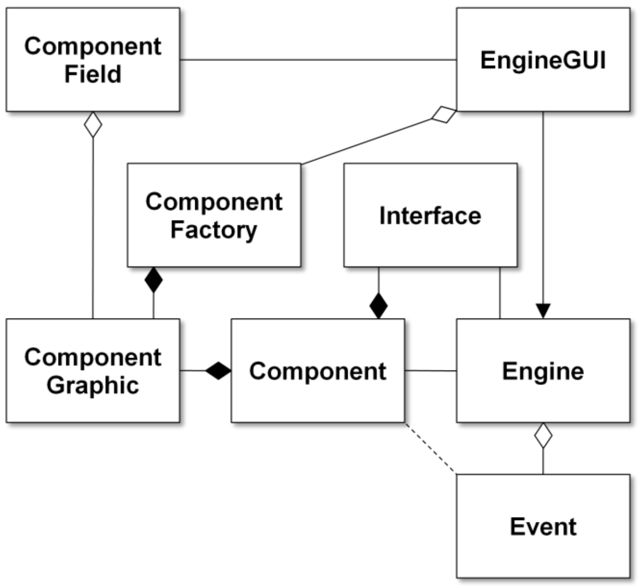


Figure 6 – The Embedded System Simulator Program Structure

An overview of the connections between each class is shown above in Figure 6 with each class explained in more detail below.

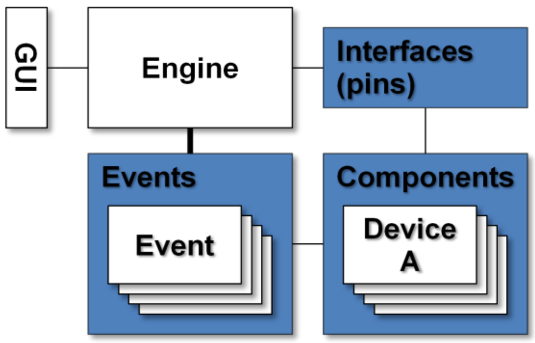


Figure 7 – Basic Architecture Overview

Figure 7 provides a more abstract view of the entire system. For the simulation description the following terminology applies:

- **Events** – actions in the system such as a toggling of an output, tick of a clock, etc.
- **Interfaces** – objects representing the external electrical interfaces of a component which are used to communicate with other components. This includes pins, pads and wires.
- **Engine** – the simulation engine which coordinates and executes all operations.
- **Components** – these objects represent electrical components such as logic gates, flip flops, and microcontrollers.

The important classes which have been created to support the design of this system are described below.

4.2 Events

Events are created by components within the system to represent actions which must be executed in the future. They consist of an `action` method and a time at which the event is to be executed. Events are typically created in the following situations:

1. When initialising the `Component` object
2. Within the execution of another `Event`
3. In response to a change in the input `Interface` values of a `Component`

The first two cases can be easily explained by considering a clock. When such a component is initialised it would need to create a `tick` event in the near future to make the clock toggle the output value. Whenever a `tick` event is executed the value of the output Interface would be changed to the opposite of what it currently is and a new `tick` event would be created for the next clock tick. In this example each `tick` event would be set to execute at the time $STime + (\frac{period}{2})$ where $STime$ represents the current simulation time and $period$ is the period between each clock tick.

The final case is best explained considering a logical NOT gate. When the input of the NOT gate changes, the output must change in response. In this case, the action of the event might be to call `setOutput` (with the new output value as a parameter) and the time of the event would be calculated as $STime + propagation\, delay$ where $STime$ is the current simulation time and $propagation\, delay$ is the time it takes for the change in input to propagate into a change of output.

All events must be sent to the simulation Engine where they are added to a priority queue and executed in order of time.

4.3 Interfaces

All device inputs and outputs (pins, wires, etc.) are represented by `Interface` objects. These `Interface` objects are responsible for calculating (when in input mode) and storing their value, keeping track of all connected Interfaces (siblings), creating a graphical representation, and notifying the simulation Engine when any changes to value or direction occur.

The value of an input Interface is currently calculated such that the value is $logical\ 1$ if and only if $\{\exists x \in I \mid x\ is\ output\ AND\ x\ is\ logical\ 1\}$ where I represents the set of the siblings to the Interface in question. This calculation can be customised by extending the Interface class and overwriting the `calculateValue` method.

Interfaces can be changed from input and output mode (and vice versa) at any time by the `Component` to which it belongs, providing support for devices with shared input and output pins.

Whenever a value or direction of an Interface changes, the simulation `Engine` is notified automatically through the use of an Observer design pattern. In an earlier version of this class the responsibility lay with the parent `Component` to notify the `Engine` of these changes however this modification has simplified the `Component` design process significantly.

A design decision also was made to limit the value range to the digital spectrum only. This simplified both system operation and component logic drastically at the cost of inability to represent other values. Additional values are useful is describing situations such as when no connection is made to an input pin, or when two competing values are supplied to and input.

4.4 Engine

The simulator `Engine` contains all the methods required for adding and removing components to the system, connecting and disconnecting Interfaces (pins) together, and adding and executing events. It also keeps track of the current simulation time.

Whenever a component is added to a system (currently done via the GUI) the engine is notified via a `subscribe` call, conversely whenever a component is removed the engine is notified by a call to `unsubscribe`. These two methods simply allow the engine to keep track of all the components within the system and to disconnect all connected `Interfaces` of a component being removed.

To connect or disconnect two interfaces (once again done via the GUI) calls are made to `connect` and `disconnect` respectively. The engine simply passes on these instructions to the two interfaces upon which the action is occurring.

Whenever a component creates a new event it must be added to the `Engine`, this is done via the `addEvent` method. All `Event` objects are stored within a `PriorityQueue`, providing a constant time `peek` operation and log time `remove` and `add` operations, with the next event in system time being at the top of the queue. The given `Event` is checked to ensure it occurs in the future and is then added to the queue.

The `start` method is one of the most interesting as it initiates and controls the simulation of the entire system. First this initialises the system and all components and then enters an infinite loop. This loop consists of a call to `tick` for each time events at the next available time should be executed, and a mechanism to make the machine run at a real-time simulation speed if necessary. Every time around this loop the system checks the state of the system, if the system has been told to pause operations the executing thread will wait until notified by an `unpause` or `unbreakpoint` call. If the system has been told to stop, the loop will exit and allow a natural completion of execution.

The `tick` method, mentioned above, is the method in which system events are executed. When called, this method first updates the engine time to that of the next event in the queue and then executes all events at the specified time. The execution of these events may cause the creation of new events to be added to the queue (e.g. a clock's tick event will enqueue the subsequent tick event in the future) and may also result values of `Interfaces` to change. The latter is recorded and all owners of connected `Interfaces` are notified of the change such they may enqueue any new events as a result. An example of this is the input to a NOT gate changing from a $logical\ 1$ to a $logical\ 0$ – this change would cause the gate to enqueue an event in the near future to change its output to $logical\ 1$, the time delay between the change in input and output represents the propagation delay of the device in this case.

4.5 Components

All components must extend the `Component` class. This ensures all components have a common set of methods which are used to interact with the other parts of the system such as the Engine, Interfaces and GUI. Components are responsible for modelling the behaviour of the simulated device, with respect to time and input values. They are also responsible for creating and updating their graphical representations by creating an object which extends `ComponentGraphic`.

Components are notified of value changes to any connected interfaces via a call to the `interfacesChanged` method. Commonly, this would cause a subsequent `Event` to be added to the system to respond to this change. It is crucial that the device not have any immediately external response to these changes – all reactions must have some (even if very minor) delay to maintain stability of the system.

All components must also implement `getInterfaces` which simply returns an array of all the interfaces which belong to the device.

When the simulation is started the Engine calls the following two methods to initialise the `Components`. The first, `prepareOutputs,` informs the component to initialise all `Interface` directions and values. The second, `init`, is called to allow the `Component` to enqueue any required starting events (e.g. the clock component's first `tick` operation).

Finally, when ending a simulation the `dispose` method is called. This method must finalise any additional operations the component performs including closing GUI windows and network sockets, or ending additional threads.

The following details the inner workings of the more interesting or complicated components. For the Javadoc for the Component class see Appendix A.

4.5.1 Logic Gates

The design of the logic gates in the ESS is an example of the ability to abstract common code between similar components (shown in Figure 8). The gates were designed by creating an abstract class which extends `Component` called `AbstractGate`.

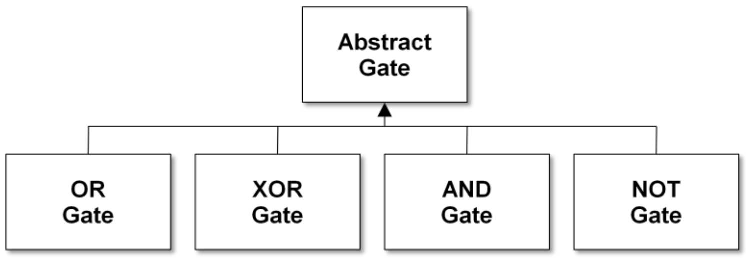


Figure 8 – Code Abstraction With Logic Gates

This allowed each individual gate to only have to provide a `refresh` method, which calculates the new output value based on current inputs. Additionally a graphical representation of the gate may also be provided if so desired, the NOT gate is the only gate that implements its own graphic.

The ability to abstract functionality in this manner is one of the main advantages of the new Embedded Systems Simulator.

4.5.2 Atmel Microcontroller

The `AT90S8515` plug-in is by far the most complicated and resource intensive component used in the simulator. The internal logic has been separated into multiple classes to ease both coding and understanding. Explanations of the separate classes are described below.

4.5.2.1 Instruction

This abstract class represents a single code instruction. Each implementation of an `Instruction` must have the methods `toString`, to convert the instruction into a human readable form, and `execute`, a method to execute the given instruction using a `Core` object. Specialised subclasses of `Instruction` include:

- `Instruction32Bit` – extended by all 32 bit instructions, this is useful in executing instructions such as SBIC and SBIS (skip the next instruction based on a bit in an I/O register) which must be able to determine the size of the next instruction. Using these objects simplifies the identification of instruction types using Java's `instanceof` statement.
- `Instruction32BitData` – this is used to represent the data segment of a 32 bit instruction and thus should never be executed. This throws an exception on an attempt to execute.
- `InstructionUnknownOrData` – this represents instructions which cannot be determined by the `Interpreter`. This could be an instruction which is unsupported by this microcontroller, an instruction which has been badly formatted, or data which is being stored in the program memory.
- `InstructionUnsupported` – this is used to represent instructions which have been correctly interpreted but are not currently supported by the Embedded System Simulator.

`Instruction` objects also allow a breakpoint value to be set to either true or false. If an `Instruction` with breakpoint set to true is the next to be executed, the AT90S8515 Component object will stop before it is executed.

4.5.2.2 Hex File Parser

This provides static methods to parse a given file, formatted in I16HEX (Intel HEX 16 bit), and return an array of bytes filled with the data contained within the file. It is important to note that this parser does not currently ensure the checksums within the file are correct. This class is used to interpret the compiled hex files into a form which can be disassembled by the `Interpreter` class.

4.5.2.3 Interpreter

The `Interpreter` class provides static methods which take byte code and disassemble it into respective `Instruction` objects. This conversion can be done either individually or as a batch operation.

Each instruction is identified using a series of `switch` and `if-else` statements and then is passed to a respective method to create the `Instruction` object. The speed of this could be improved using a lookup table for each instruction (the size being 2^{16}) however as this was deemed unnecessary as identification only occurs once while loading each chip. In the future this may need to be extended to provide sufficient speeds for microprocessors supporting the SPM (store to program memory) instruction in which instruction byte codes must be interpreted during simulation.

4.5.2.4 Data Memory

This object is used to handle all operations. This object is responsible for reading and writing to memory locations, as to enforcing special rules for specific memory locations which behave differently to others. Examples include:

- The UDR memory location which reads and writes to two different registers using one address. When the UDR memory register is read a flag must be cleared in the USR register.
- The flag registers (GIFR, TIFR, etc.) which, when written to, actually perform a bitwise NOT AND operation between the current and the given value.

The `DataMemory` object also provides the ability for classes to register as “listeners” to individual memory locations. This allows the classes to be notified when the given memory location value is modified. This mechanism allows other parts of the system to operate efficiently, not having to check the register values upon each clock tick.

Additionally, one object at a time may register to listen for changes to the interrupt flag in the SREG (status register). This special case was designed to decrease the overhead caused notifying the `Interrupt` subsystem upon each SREG change as this register is likely to change value very often.

An object must implement the `DataMemoryListener` interface to be eligible to subscribe any of the notifications describes above. This design is an implementation of the Observer software pattern [24]. The use of the Observer pattern allows other parts of the system to be simplified drastically. If this pattern had not been used then either each subsystem (Timers, UART, Port, and Interrupts) would have to recheck each respective control registers or any action affecting the value of registers would have to check to determine if the change made affects any of the subsystems.

4.5.2.5 AT90S8515Core

This object holds the array of `Instructions` for a given chip, the current program counter, and provides the methods required for executing all instructions. This object was abstracted from the main component class due to the large amount of code required to implement each instruction.

4.5.2.6 Port

This class is used to represent each I/O ports of the microcontroller. It groups all port operations together based on their respective registers (direction and output) and the values of the `Interface` objects for each pin. Each `Port` monitors changes to its direction and output registers through the use of the `DataMemoryListener` interface and the ability to subscribe to changes to given registers through the `DataMemory` object. Using this method avoids having to check each clock cycle for an update to each port register, increasing the achievable speed drastically. Each port is notified of an input change via the main `Component` object when the `interfacesChanged` method is called.

4.5.2.7 Timers

Both timers of the microchip are represented using a single `Timers` object. The `Timers` object keeps track of the state of the timer control register and the timer compare registers using the `DataMemoryListener` interface. The `update` method must be called each clock cycle to increment then as necessary. When a timer is incremented, the timer values are changed according to their control registers (clear on compare match, etc.) and any respective flags are set or cleared (overflow, compare match, etc.).

If, according to the timer pre-scale, the timer should not be incremented in the given clock cycle, the update call does very little execution and returns almost immediately.

4.5.2.8 UART

The `UART` object contains the logic required for the Universal Asynchronous Receiver-Transmitter part of the microchip. This object monitors all registers relating to the UART, using the `DataMemoryListener` interface, and handles the transfer of data in and out of the chip via the registers and the `Interfaces`. The object contains two variables for storing the shift in and out register. All behaviour has been designed to be identical to that of the actual chip.

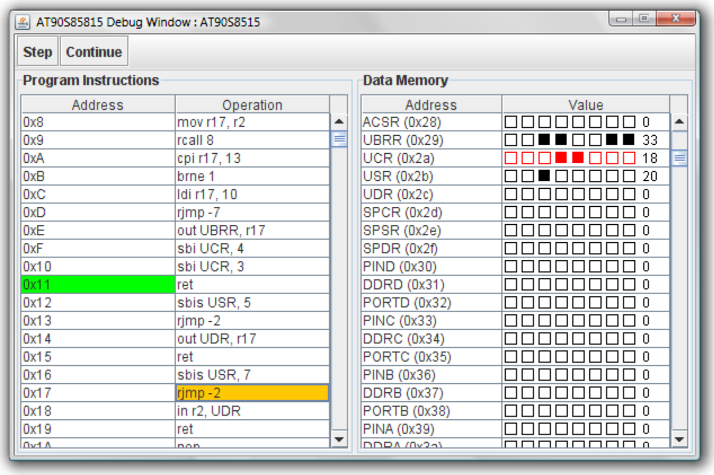
4.5.2.9 Interrupts

The microchip interrupts are represented using the `Interrupts` object. This object monitors all interrupt control and flag registers using the `DataMemoryListener` interface, it also utilises the ability to monitor just the interrupt bit in the status register (described earlier in 4.5.2.4). The abstraction of this object allows the timers, UART and other subcomponents delegate the responsibility of interrupts and focus on their individual operating and flag setting/clearing.

A minor complication arose from the fact that each interrupt takes one clock cycle to trigger. Due to this the `Interrupt` object must be "updated" every clock tick to execute any interrupts identified during the last clock cycle. This call executes in very little time if an interrupt has not been triggered as it only has to check one flag and then reurn. If an interrupt has triggered the respective actions are undertaken (push the program counter on the stack, clear the interrupt flag, and update the program counter to the triggered interrupt vector).

4.5.2.10 Debug Window

The debug window is a simple class which extends `JFrame` and creates two tables within scroll panes (seen in the background of Figure 9). Each table implements a specialised `TableCellRenderer` which is responsible for highlighting the current instruction of execution, the instructions which have breakpoints set, and the memory cells which have been modified since the last breakpoint. The `DebugWindow` also implements a listener to monitor for double clicks on the instruction table and set breakpoints if such an event occurs. The majority of the functionality within this graphical display was implemented using the Java swing libraries.


AT90S85815 Debug Window : AT90S8515

Step | Continue

Program Instructions

Address	Operation
0x8	mov r17, r2
0x9	rcall 8
0xA	cpi r17, 13
0xB	brne 1
0xC	ldi r17, 10
0xD	rjmp -7
0xE	out UBRR, r17
0xF	sbi UCR, 4
0x10	sbi UCR, 3
0x11	ret
0x12	sbis USR, 5
0x13	rjmp -2
0x14	out UDR, r17
0x15	ret
0x16	sbis USR, 7
0x17	rjmp -2
0x18	in r2, UDR
0x19	ret
0x1A	nop

Data Memory

Address	Value
ACSR (0x28)	□□□□□□□□ 0
UBRR (0x29)	□□■■□□■■ 33
UCR (0x2a)	□□□■■□□□ 18
USR (0x2b)	□□■□□□□□ 20
UDR (0x2c)	□□□□□□□□ 0
SPCR (0x2d)	□□□□□□□□ 0
SPSR (0x2e)	□□□□□□□□ 0
SPDR (0x2f)	□□□□□□□□ 0
PIND (0x30)	□□□□□□□□ 0
DDRD (0x31)	□□□□□□□□ 0
PORTD (0x32)	□□□□□□□□ 0
PINC (0x33)	□□□□□□□□ 0
DDRC (0x34)	□□□□□□□□ 0
PORTC (0x35)	□□□□□□□□ 0
PINB (0x36)	□□□□□□□□ 0
DDRB (0x37)	□□□□□□□□ 0
PORTB (0x38)	□□□□□□□□ 0
PINA (0x39)	□□□□□□□□ 0
DDRA (0x3a)	□□□□□□□□ 0

Figure 9 - The AT90S8515 DebugWindow

4.5.3 UART to TCP

A large portion of the code in the `UARTtoTCP` component is identical to the UART on the AT90S8515, however the `Component` must also create a new network socket to listen for incoming data. An additional thread (started on component initialisation) creates the network socket is listens with the supplied port number, accepts any incoming connection, routes all information gathered from port out of the component via the "transmit" `Interface`, and transmits all received information from the receive `Interface` to the network connection. This allows applications like HyperTerminal to be used in a similar way as they would if communicating with a real microcontroller.

As the component simulates UART communication using the `Interfaces`, if the baud rate of the sender and receiver components do not match, the received data received will be garbled. Additionally, when the `UARTtoTCP` component detects an error in transmission (detected when the stop bit is not a *logical* 1) the respective `ComponentGraphic` will change colour to notify the user of the error.

It is important to note that this component does not support 9 bit UART streams. This was not done as the byte stream being used with UART does not support values other than 8 bits.

4.5.4 Delay

This component is relatively simple with only one input and one output, however it highlights a very useful feature of the Embedded System Simulator. Whenever the Delay component is notified of a change in value of its input Interface it immediately adds an event to change update its output value to be the same to be executed at some time in the future. This time delay is specified when creating the component and can range from less than 1 yoctosecond to well beyond a millennia.

4.5.5 Complete List of Components

A complete list of components currently included with the Embedded System Simulator:

- D Flip Flop
- JK Flip Flop
- SR Flip Flop
- T Flip Flop
- Push Button
- AND Gate
- OR Gate
- XOR Gate
- Not Gate
- Clock
- AT90S8515
- UART to TCP
- Delay
- Ground

4.6 Component Graphic

The `ComponentGraphic` class is an abstract class which must be extended by all graphical representations of components. This class provides some base functionality, to reduce code repetition, and allows the `ComponentField` identify and store these graphics easily.

These are extended both for the graphical representation of the component when placed in the `ComponentField` and also the icon used when placing the component under the cursor. These are supplied by the `Component` and `ComponentFactory` objects respectively.

4.7 Component Factory

`ComponentFactory` is an abstract class which must be extended for each possible `Component` which can be created. This class is responsible for storing the name, the graphic icon, and the list of configurations options required to instantiate their respective `Component`. The configuration options are retrieved through the method `getSettings` method, which returns an array of objects implementing the `Setting` interface (see Section 4.8 for more information).

The `ComponentFactory` class is modelled as an abstract factory using the Abstract Factory design pattern [24]. Each implementation of `ComponentFactory` must contain constructor which takes no parameters. `ComponentFactory` contains the required information to instantiate their respective `Component` objects, hiding the specifics from the rest of the system. This architecture is used to allow components to be created with settings sent to them via their constructor methods, while also hiding these specifics from the `EngineGUI`. When the ESS initiates, it uses reflection to search for non-abstract implementations of `ComponentFactory`, instantiate, and create a tree icon for each.

For the Javadoc for the ComponentFactory class see Appendix B.

4.8 Setting

The `Setting` class is an interface which all specific settings must implement. This ensures methods for retrieving the name of the setting (e.g. "Clock Speed", "Number of Inputs", etc.) and the level of complexity (necessary/normal/advanced setting) are all available from a `Setting` class.

These classes allow `ComponentFactory` objects to specify a list of options for their respective `Component`.

Classes which currently extend `Setting`:

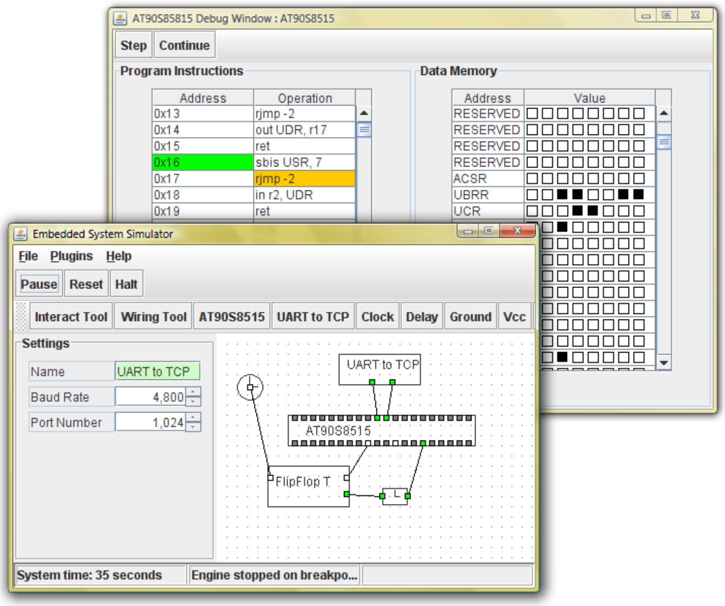
- `StringSetting`
- `IntegerSetting`
- `DoubleSetting`
- `FileSetting`

4.9 Engine GUI

The `EngineGUI` object is a simple wrapper class for `Engine` which contains and updates the `ComponentField` object by catching all important method calls in the `Engine`. This object is also currently responsible for locating all non-abstract classes which extend `ComponentFactory`. Upon creation these classes are located and each object is created using reflection, allowing an icon to be created for each factory object. When this icon is selected the respective `Settings` are retrieved from the object and loaded into the settings frame, and the `ComponentField` is sent a copy of the `ComponentFactory`.

4.10 Component Field

This object's sole purpose is to display the graphical representation of a circuit board (see Figure 10). It allows `Components` to be placed on the board at given positions and provides mechanisms for connected these `Components` together by clicking on the `Interfaces`. The `ComponentField` currently only provides rudimentary connections, drawn as straight black lines between the connected interfaces. While this does not affect the usability of the system in any adverse way, it does reduce the professional appearance and crowd more populated circuits.



Address	Operation
0x13	rjmp -2
0x14	out UDR, r17
0x15	ret
0x16	sbis USR, 7
0x17	rjmp -2
0x18	in r2, UDR
0x19	ret

Address	Value
RESERVED	□□□□□□□□
RESERVED	□□□□□□□□
RESERVED	□□□□□□□□
RESERVED	□□□□□□□□
ACSR	□□□□□□□□
UBRR	□□■■□□■■
UCR	□□□■■□□□

Name	UART to TCP
Baud Rate	4,800
Port Number	1,024

System time: 35 seconds

Engine stopped on breakpo...

Figure 10 - Screenshot of the New Embedded System Simulator

When the `ComponentField` is sent the selected `ComponentFactory` object by the `EngineGUI` and the user moves the mouse over the `ComponentField` GUI the respective `ComponentGraphic` is added and positioned under the cursor. If the user then clicks the mouse, the `ComponentFactory` is told to create a new instance of its respective `Component`. This new `Component` object is added to the `Engine` and placed on `ComponentField` in the position defined.

The `ComponentField` object contains a timer which continually repaints the entire user interface (`Components`, `Interfaces` and wires) every 20 milliseconds. This task was originally the responsibility of the `Interface` and `Component` objects however it was delegated due to extreme situations in which `Interface` objects may change value 2 million times per second. This allows the simulation to display the latest information while also allowing real time simulation even in extreme circumstances at the cost of extra CPU usage in normal situations. It was decided this was an acceptable trade-off as CPU usage during normal situations is lower, and as such expendable.

4.11 Embedded System Simulator

This contains the main method for the Embedded Systems Simulator. When executed, this method creates a new instance of the `EmbeddedSystemSimulator` object which in turn creates the main GUI window (see Figure 10). This window contains the basic buttons to control the simulation (play/pause, stop, and reset), and the status fields which display the current information about the system (the state and time).

When the control buttons are pressed the system checks a local variable to determine if the `EngineGUI` object has been created. If it has it is sent the respective method, otherwise if the play button was pressed the system creates a new `EngineGUI` within a new `Thread`, which in turn is executed. This allows the main GUI to remain responsive to user input while the simulation is executed in the background.

The `EmbeddedSystemSimulator` also contains a timer by which the `EngineGUI` is regularly polled and status fields updated. The first field simply displays the current time passed within the `Engine`, the second displays a comparison between simulation and real-time. This is calculated by measuring comparing the differences in the simulation and real-time between consecutive updates.

5 Evaluation of Project Outcomes

This chapter compares and reviews the differences between the thesis specification, in Chapter 3, against the actual implementation described, in Chapter 4. The correctness and completeness of the design will be analysed and as such the achieved levels usability, usefulness, and design quality will be assessed. Finally, the process undertaken to develop this thesis will be described.

5.1 Correctness

This section details the methods by which the Embedded System Simulator and accompanying components were tested and the results were gathered.

The implemented testing method was iterative, progressive, system-wide testing focusing on the Engine and the AT90S8515 component due to their complexity relative to other sub systems.

The Engine was tested upon each new significant development in any component or other part of the system. To test the Engine, components specially crafted to test the area recently changed were added and the simulation started. The behaviour of that system was compared to the expected behaviour and any identified errors were either fixed immediately or documented in the source code using the "TODO" function in Eclipse. The advantages of this technique was a quick feedback cycle, however a significant disadvantage was the lack of systematic testing. This allowed bugs which may develop in previously tested components to go unnoticed.

In order to test the Atmel microcontroller, it was important to use code which was well know to avoid errors from the simulator or code from disguising themselves as errors from each other. Additionally the behaviour of this code must be previously well known.

The code chosen to test the Atmel component was selected from the same code for which the simulator was originally created – the Introduction to Computer Systems practical examples and answers. This code was available in both ASM and C versions, allowing for a broader and more gradual testing process.

The tests were implemented in the same order as the practicals as this offered a gradual scale in the difficulty of the instructions used. Due to the nature of compiled C files, these were left to last, retesting the overall behaviour with larger code bases, as well focusing on the load, store operations. Overall these files provided an excellent test bed for this plug-in as they progressively tested new features of the chip as they were created.

After all the previous code was tested and behaviour verified to be correct, the code for the final project in the Introduction to Computer Systems course was loaded into the simulator. Unfortunately this did highlight some additional errors, some of which were corrected but others which were not. These remaining errors are highlighted in the next section.

The final code that was executed within the system was a "stress test". This "stress test" was written to do the most computationally intensive tasks to ensure the engine was capable of running at real-time even under the most intense situations. This code did the following operations simultaneously:

- Ran both Atmel timers
- Sent and received data via UART
- Toggled all unused outputs every four microcontroller cycles

This test allowed continual checking on the systems performance. Whenever running this test highlighted a large change in efficiency the latest change was reviewed and optimised. This test helped identify many slow operations while designing the system. When testing the interrupts section of the AT90S8515 component, it was identified that adding an observer to changes in the status register slowed the system down below acceptable levels. This prompted the `DataMemory` object to be rewritten, providing the ability of watching for changes in just the Interrupt flag. This modification restored the maximum speed of the simulator to previous levels.

5.2 Completeness

This section compares each requirement specified in section 3.2 with the actual implementation and progress made.

5.2.1 Software Architecture

The software was designed using a plug-in architecture in mind, this however was not fully achieved. The aim was for the system was to search in a folder for non-abstract classes (within jar packages) which extend the `ComponentFactory` class. However, currently the system is programmed such that it looks at all classes specified within an array of Strings as opposed to a predefined folder. As this requires the class to be packaged with the original program it requires rebuilding each time a new component is created.

5.2.2 Required Components

Following is the list of required components coupled with a description of their status within the Embedded System Simulator.

- Push Button/Switch
  This component was implemented successfully however it does not provide the ability to simulate contact bounce.
- Light Emitted Diode (LED)
  The graphical representation of `Interface` objects automatically added functionality similar to what was required of the LED, removing the need for an LED component.
- Atmel AT90S8515 Microcontroller
  This component was the most complicated of all developed. It is able to load, disassemble, and execute compiled HEX files. It also provides the ability to view the internal state, add breakpoints, and step through code. It does not currently support:
  EEPROM Memory
  - Analogue Comparator
  - Serial Programmable Interface (SPI)
  - Watch Dog Timer
  - Low Power/Idle Modes
  - External interrupts
  - Pulse Width Modulation Timer Mode
  - External clocking ability

  Unfortunately there are also some errors which remain in the instruction interpretation or executor. These errors do not affect the operation of the practical examples from the Introduction to Computer System's course however the simulator is unable to correctly execute the code from the final project.

- MAX232 Chip
  This has been renamed to “UART to TCP Component” to better describe it’s functionality. This operates completely as desired, receiving and sending data via UART and a TCP port. It also replicates errors due to incorrect baud rate in practically the same way the hardware counterpart would.
- Seven Segment Display
  Unfortunately this was not completed due to time constraints, however it is a simple component which could easily be added to the system.
- Piezoelectric Speaker
  Due to restraints in the Java programming language this was not implemented. An alternative component is suggested in section 5.6.2.

Additional component were however created to emphasise the wide range of uses of the new Embedded Systems Simulator:

- Flip Flops (JK, D, T, SR)
- Logic Gates (AND, NOT, OR, XOR)
- VCC and Ground
- Clock
- Delay

For more information regarding these new components see Section 4.

5.2.3 Graphical User Interface

The overall system was design such that the graphical user interface was abstracted from the rest of the system, hopefully allowing reuse in future projects. Unfortunately the ability to combine components together into reusable sets is not available in the new Embedded System Simulator.

5.2.4 Performance

The simulator is capable of running a simulation, consisting of an AT90S8514 microcontroller plug-in (running at 4MHz) and many accompanying components, at real-time speed using approximately 20% CPU usage on a 2GHz dual-processor laptop running Windows Vista. This meets and exceeds the performance requirements.

5.3 Design Quality

The design of the class structure was carefully considered prior to starting the actual coding. All required classes and connections were described in both a specification document and graphically using a basic form of UML notation. Many design patterns were used in package and class design to help improve the quality, an example of this is the Abstract Factory pattern described in section 4.7. This process helped ease the design of the entire system and improved the quality of the final product.

Some refactoring was done during the design phase as new complications arose and issues were identified. A specific example of a refactoring done to improve code quality and ease the development of new Component objects was to move the responsibility of notifying the Engine of any changes to Interfaces from the parent Component to the Interface objects themselves. This was done using an Observer pattern.

5.4 Usability and Usefulness

The final thesis product's usability should be evaluated in terms of its initial target audience, the Introduction to Computer Systems university course. The main objective was for the simulator to be capable of real-time simulation of both the practical examples and the final project.

5.4.1 Practicals

The new Embedded Systems Simulator is able to simulate all the practical coding examples, written in C and ASM, without errors. Unfortunately, two hardware pieces are not available for these practicals – the piezo electric speaker and the seven segment display. Without these two components the practical cannot be fully experienced by the students.

5.4.2 Final Project

It is assumed that the final project would require specialised hardware, such as the LED matrix used in the project of semester 2 2007, which would not be developed within this thesis. Such hardware components would have to be designed prior to the project.

Unfortunately, due to some minor errors in the instruction interpreter and executor, the simulation of the AT90S8515 is not currently capable of running the final project without errors. These software bugs must be located and corrected prior to being used by students for this task.

5.5 Time Management

The Progress Report contained a progress plan and Gantt chart (See Appendix C) for the two semesters over which the thesis was to be completed. Each task was accompanied with a risk analysis and likely affects should any of the risks occur.

The plan was carefully followed over the first semester. All tasks were started on time with the exception of the Atmel plug-in. It was discovered that little work could be done on the Atmel plug-in until the simulation engine was in working order.

In building the simulation engine tests were needed to ensure its correctness. It was decided that some additional components, flip flops and logic gates, would have to be made to test the basic operations of the simulation. Additionally, a GUI was needed to be able to add components to the system and observe their behaviour. These unforseen tasks delayed the overall schedule such that all tasks after the creation of the simulation engine were delayed by two months.

Progress was pushed back to the start of semester 2. During the first month a large amount of work was put into both completing the simulator ready for testing the new components and the construction of the Atmel plug-in. After this month the system was in a suitable enough condition to be able to dedicate 100% of energy elsewhere. Unfortunately, due to additional university responsibilities, all thesis work was once again neglected. Towards the end of the semester, three weeks prior the demo, work began once again at a much faster rate. More hours in fewer days were spent working on getting the components finished, the user interface polished, and bug testing done. During the final week of preparation, the number of hours spent working on thesis reached 90.

5.5.1 Analysis of Time Management

The level of time management during the overall project was very lacking, as was the initial plan. Prior to starting work the plan should have been thought out much better and broken down into smaller, individual tasks. This would have allowed a much more structured schedule and provided an excellent base of comparison against actual progress.

Additionally, had I made regular comparisons between progress made and the initial schedule I would have noticed much earlier the lacking progress.

5.6 Future Work

Many improvements could be made to the underlying logic, performance, and graphical user interface of the new Embedded Systems Simulator. These modifications and extensions have been listed below.

5.6.1 Graphical User Interface

The graphical user interface currently contains some minor problems. The first and foremost is the fact that the `ComponentField` object repaints the entire display 50 times per second regardless of any changes to the state of the simulation. An improvement would be to make the GUI only update the required every 20ms if a change has occurred in the system (i.e. an `Interface` changing direction or value). It is important to note that the GUI should not be updated upon every change as this would cause excessive CPU usage in certain situations (such as the stress test described in Section 5.1).

Currently the `ComponentField` also offers very little functionality in terms of laying out the components and connections. Once a component has been added it is impossible to change its position or remove it from the system without removing all other components at the same time. It is also possible to place two components in the exact same position on the board. Connections between Interface objects are shown as straight lines – this could be extended to allow users to define the paths and customisable colours. An example of what these changes may look like is shown in Figure 11.

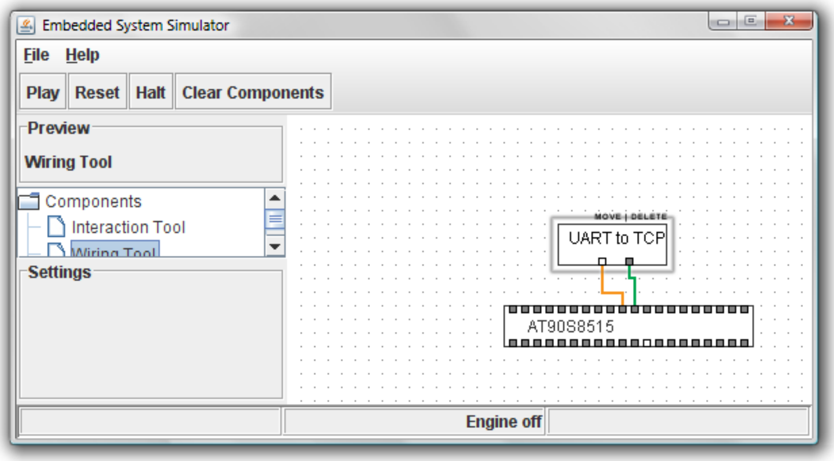


Figure 11 – Example of a New GUI With New Features

5.6.2 Additional Components

Many more components could be added to the current collection. Some examples of possible components are listed below in order of their approximated value.

- Seven Segment Display
- Piezoelectric speaker – this may not be possible by Java, instead a Component may be created with one input which analyses the frequency displays the note of the sound which would be generated
- Liquid Crystal Display (LCD) Unit
- Keypad

Finally, additional Atmel microcontroller models could be added to the system, extending the AT90S8515's code.

5.6.3 Performance

The performance of the system should be analysed using a profiler. This would allow the slower sections of the application to be identified and optimised, allowing more intensive simulations to take place (i.e. simulations with more than one Atmel microcontroller).

5.6.4 Additional Functionality

The features of the Atmel debugging environment could be greatly extended. Many alternative debugging environments provide functionality such as watch points, additional stepping options, and symbol level debugging. All of these could be implemented in the AT90S8515 component fairly easily.

It would also be useful to be able to save component configurations, and possibly states, for distribution or loading at a later date. This would require the ability to store a unique identifier for each existing (and future) `Component` objects.

Products reviewed in section 2.2, specifically AVR Studio, provide Integrated Development Environments (IDE) including a compiler, programmer, and simulator all in one. The Embedded Systems Simulator could be extended to include these other features, providing a much better environment to program for these platforms. This is however a long term goal as these features are complicated and difficult to do in the Java programming language.

6 Conclusion

This document has covered the research, the specification of the project, the design process, and analysis of completion. The primary design decisions made during the development were highlighted and explained.

The overall completeness of the project leaves much to be desired. The current application is in a form which is mostly usable to the target students however work still needs to be done before it reaches a state which can be distributed. Many tasks have been suggested for future implementation, these both complete and improve the current product provided.

The desired goals of this project were very ambitious and while they have not all been completed the final product is a large improvement over previous version of the Embedded System Simulator. The simulator is capable of real-time simulation of the Atmel chip specified and additional Components. A dissembler and debugger has been created which allows users to view the entire internal state and control the flow of execution of the AT90S8515 microcontroller.

Had more time been available or better time management practices been implemented the final small issues with the system could have been resolved and the application distributed. Work is planned to continue beyond the period of this thesis. A version meeting all the original desired attributes, will be completed and supplied to the university within the next few months.

Bibliography

[1] J. Banks, J. S. Carson II, B. L. Nelson, *Discrete-Event System Simulation*, Prentice-Hall, 1996.

[2] “Supported System Configurations”, Sun Microsystems, http://java.sun.com/j2se/1.5.0/system-configurations.html, 2007

[3] Atmel Corp. "AT90S8515 Product Document", http://www.atmel.com/atmel/acrobat/doc0841.pdf, September 2001

[4] J. Banks, *Handbook Of Simulation*, John Wiley & Sons, 1998

[5] T. J. Schriber and D. T. Brunner, “Inside discrete-event simulation software: how it works and why it matters”, Simulation Conference Proceedings, 1999, pp. 72-80

[6] Application Note 1796, “Overview of 1-Wire Technology and Its Use”, http://pdfserv.maxim-ic.com/en/an/AN1796.pdf, December 2003

[7] S. H. Gerez, *Algorithms for VLSI Design Automation*, John Wiley & Sons, 1999

[8] D. A. Tacconi and F. L. Lewis, “A new matrix model for discrete event systems: application to simulation”, IEEE Control Systems Magazine, vol. 17, pp. 62-71, October 1997.

[9] H. Muhr and R. Holler, “Accelerating RTL Simulation by Several Orders of Magnitude Using Clock Suppression”, Embedded Computer Systems: Architectures Modeling and Simulation, July 2006, pp. 123-128

[10] G. A. Korn and J. V. Wait, *Digital Continuous-System Simulation*, Prentice-Hall, 1978

[11] J. C. Comfort, “The simulation of microprocessor based event set processor”, Proceedings of the 14th annual symposium on Simulation, 1981, pp. 17-33

[12] L. Gauthier and A. A. Jerraya, "Cycle-true simulation of the ST10 microcontroller including the core and the peripherals," SLS group, TIMA Laboratory, Grenoble, France 2000.

[13] F. Howell and R. McNab, “simjava: A Discrete Event Simulation Library For Java”, citeseer.ist.psu.edu/howell98simjava.html, 1998

[14] R. McNab and F.W. Howell, " Using Java for Discrete Event Simulation", Twelfth UK Computer and Telecommunications Performance Engineering Workshop (UKPEW), Univ. of Edinburgh, 1998, pp. 219-228

[15] R. A. Kilgore, K. J. Healy and G.B. Kleindorfer, “The future of Java-based simulation”, Simulation Conference Proceedings, December 1998, pp. 1707-1712

[16] K. J. Healy and R. A. Kilgore, “Silk : A Java-based Process Simulation Language”, Simulation Conference, December 1997, pp. 475-482

[17] R. A. Cassel and M. Pidd, “Distributed Discrete Event Simulation Using The Three-Phase Approach And Java”, citeseer.ist.psu.edu/266379.html, 1999

[18] R.A. Kilgore, “Open-source SML and Silk for Java-based, object-oriented simulation”, Simulation Conference, 2001, pp. 262-268

[19] P. A. Fishwick, “Object-oriented simulation”, ACM SIGSIM Simulation Digest, June 1988, pp. 19

[20] A. Schwarz , “Simulieren und Debuggen mit SimulAVR, GDB, Insight, AVRStudio”, http://www.mikrocontroller.net/articles/AVR-Simulation or http://64.233.179.104/translate_c?hl=en&sl=de&u=http://www.mikrocontroller.net/articl es/AVR-Simulation, 2004

[21] D. Goh, “Embedded System Simulator”, Bachelor of Engineering Honours Thesis, University of Queensland, Queensland, Australia, 2006.

[22] J. Kehl, “Embedded System Simulator”, Bachelor of Engineering Honours Thesis, University of Queensland, Queensland, Australia, 2004.

[23] D. Nixon, “Embedded System Simulator”, Bachelor of Engineering Honours Thesis, University of Queensland, Queensland, Australia, 2004.

[24] F. Buschmann, *Pattern-oriented software architecture*, Chichester and Wiley, 1996

Appendix A – Javadoc for the Component class

simulator.component

Class Component

```
java.lang.Object
  └ simulator.component.Component
```

`public abstract class` **`Component`**
`extends java.lang.Object`

This class represents a component in the system.

Author:

Nick

Field Summary

`protected simulator.Engine`	**`engine`** The engine this component belongs to

Constructor Summary

`protected`	**`Component`**`(java.lang.String name, simulator.Engine engine)` Create a new Component.

Method Summary

`abstract void`	**`dispose`**`()` Called to dispose of the component.
`java.lang.String`	**`getClassName`**`()` Get the class name
`abstract ComponentGraphic`	**`getComponentGraphic`**`()` Returns the graphical representation of this component
`java.lang.String`	**`getFullDescription`**`()` Print the state of the component in the following form (or as close as possible) Component: name:type Inputs: [name:value{, name:value}] Outputs: [name:value{, name:value}]
`simulator.component.Interface`	**`getInterface`**`(java.lang.String name)` Returns the interface belonging to this component with the name supplied.
`abstract simulator.component.Interface[]`	**`getInterfaces`**`()` Gets all Interface objects which belong to this component.
`java.lang.String`	**`getName`**`()` Get the name of the component.
`abstract void`	**`init`**`()` This is called to initialise the component.
`abstract void`	**`interfaceChanged`**`(java.util.Set<simulator.component.I`

	`nterface> changedInterfaces)` This is called to notify the component that inputs have changed value.
`abstract void`	**`prepareOutputs`**`()` Set all initial values for all outputs
`java.lang.String`	**`toString`**`()` Get the description of this component

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Field Detail

6.1.1 engine

```
protected simulator.Engine engine
```

The engine this component belongs to

Constructor Detail

6.1.2 Component

```
protected Component(java.lang.String name,
                    simulator.Engine engine)
```

Create a new Component.

Parameters:

`name` - The name of the component

`engine` - The engine to which this component belongs.

Method Detail

6.1.3 getName

```
public java.lang.String getName()
```

Get the name of the component.

Returns:

the name of the component.

6.1.4 getClassName

```
public java.lang.String getClassName()
```

Get the class name

Returns:

the name of the component class

6.1.5 interfaceChanged

```
public abstract void
interfaceChanged(java.util.Set<simulator.component.Interface> changedInterf
aces)
```

This is called to notify the component that inputs have changed value. When this is called respective actions should be added to the engine queue to change this component's state. The time at which these actions should occur is the current time + propagation delay for the action.

Parameters:

`changedInterfaces` -

6.1.6 getInterfaces

`public abstract simulator.component.Interface[] getInterfaces()`

Gets all Interface objects which belong to this component.

Returns:

An array of all interface objects belonging to this component.

6.1.7 prepareOutputs

`public abstract void prepareOutputs()`

Set all initial values for all outputs

6.1.8 init

`public abstract void init()`

This is called to initialise the component. This function must enqueue any starting events.

6.1.9 dispose

`public abstract void dispose()`

Called to dispose of the component. When this is called any additional windows must be closed, actions removed, etc.

6.1.10 toString

`public java.lang.String toString()`

Get the description of this component

Overrides:

`toString` in class `java.lang.Object`

Returns:

the string representing this component

6.1.11 getFullDescription

`public java.lang.String getFullDescription()`

Print the state of the component in the following form (or as close as possible)

Component: name:type

Inputs: [name:value{, name:value}]

Outputs: [name:value{, name:value}]

Returns:

A more complete description of the component and all Interfaces.

6.1.12 getInterface

`public simulator.component.Interface getInterface(java.lang.String name)`

Returns the interface belonging to this component with the name supplied.

Parameters:

`name` - the name of the interface requested

Returns:

the interface requested if it exists, otherwise null

6.1.13 getComponentGraphic

`public abstract ComponentGraphic getComponentGraphic()`

Returns the graphical representation of this component

Returns:

a ComponentGraphic representing this component and its interfaces

Appendix B – Javadoc for the ComponentFactory class

simulator.component

Class ComponentFactory

```
java.lang.Object
  └ simulator.component.ComponentFactory
```

`public abstract class` **`ComponentFactory`**
`extends java.lang.Object`

This ComponentFactory class must be extended for each component to be added to the system. It provides a method to instantiate Component objects with specific, pre-defined settings.

Author:
Nick

Field Summary

`protected simulator.Engine`	**`engine`**

Constructor Summary

`protected`	**`ComponentFactory`**`(java.lang.String name, java.lang.String path, simulator.Engine engine)` The default constructor for the ComponentFactory

Method Summary

`abstract Component`	**`createComponent`**`()` Create a new Component object for which this factory is made
`abstract simulator.component.ComponentGraphic`	**`getComponentGraphic`**`()` Get the component graphic
`java.lang.String`	**`getName`**`()` Get the name of the component
`abstract simulator.settings.Setting[]`	**`getSettings`**`()` Returns an array of all the Settings applicable for this Component
`java.lang.String`	**`getTreePath`**`()` Get the path of the component
`java.lang.String`	**`toString`**`()`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Field Detail

6.1.14 engine

```
protected simulator.Engine engine
```

Constructor Detail

6.1.15 ComponentFactory

```
protected ComponentFactory(java.lang.String name,
                           java.lang.String path,
                           simulator.Engine engine)
```

The default constructor for the ComponentFactory

Parameters:

`name` - The name of the component

`path` - The path to be displayed under, delimited by the '.' character

`engine` - The simulation engine to which this component belongs

Method Detail

6.1.16 getName

```
public java.lang.String getName()
```

Get the name of the component

Returns:

the name of the component

6.1.17 getTreePath

```
public java.lang.String getTreePath()
```

Get the path of the component

Returns:

the path of the component delimited by the '.' character

6.1.18 getComponentGraphic

```
public abstract simulator.component.ComponentGraphic getComponentGraphic()
```

Get the component graphic

Returns:

the graphical representation of the component to use while placing on a ComponentField

---

6.1.19 createComponent

```
public abstract Component createComponent()
                                   throws
simulator.component.CannotCreateComponentException
```

Create a new Component object for which this factory is made

Returns:

A new Component which was created based on the Settings gathered (if applicable)

Throws:

`simulator.component.CannotCreateComponentException` - This is thrown if the new Component could not be created.

6.1.20 getSettings

```
public abstract simulator.settings.Setting[] getSettings()
```

Returns:

an array of all the Settings applicable for this Component

6.1.21 toString

`public java.lang.String` **`toString`**`()`

Overrides:

`toString` in class `java.lang.Object`

Appendix C – Gantt Chart

Semester 1 and Holiday

Activity Name		Weeks	1	2	3	4	5	6	M	7	8	9	10	11	12	13	R	Exam		Break			
1	Preliminary Research	6		🟥	🟥	🟥	🟥	🟥	🟥														
2	Software Specification Document	2						🟥	🟥														
3	Create Progress Report	2							🟥	🟥													
4	Hand In Progress Report	-								x													
5	Design and Create Simulator Logic	8									🟥	🟥	🟥	🟥	🟥	🟥				🟥	🟥		
6	Design and Create Atmel Chip Plug-in	-																					
6.1	Logic and Operations	5										🟥	🟥	🟥	🟥	🟥							
6.2	Debugging functionality	4																		🟥	🟥	🟥	🟥
7	Seminar Preparation	2										🟥	🟥										
8	Seminar	-												x									
9	Design and Create Peripheral Plug-ins	5																				🟥	🟥

Semester 2

Activity Name		Weeks	1	2	3	4	5	6	7	8	9	M	10	11	12	13
9	Design and Create Peripheral Plug-ins	5	🟥	🟥	🟥											
10	Test System and Peripherals	5		🟥	🟥	🟥	🟥	🟥								
11	Distribute For External Testing	4				🟥	🟥	🟥	🟥							
12	Design and Create MAX232 Plug-in	2			🟥	🟥										
13	"Waterfall" Cycle Iteration	3							🟥	🟥	🟥					
14	Prepare Poster and Abstract	2										🟥	🟥			
15	Poster Hand In	-										x				
16	Oral Presentation/Demonstration	-											x			
17	Thesis Preparation	6									🟥	🟥	🟥	🟥	🟥	🟥
18	Thesis Hand In	-														x

R: university revision period, M: mid-semester break, Exam: university exam period, Break: winter holiday period