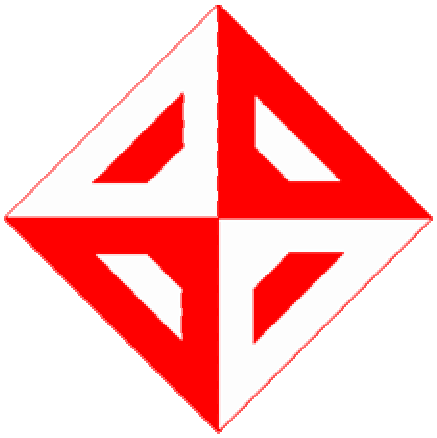
FINAL DESIGN REPORT



DEVEMB

by

ResolveSOFT

Fatih Mehmet DOĞU

Hayri Çağlayan ERDENER

Adem HALİMOĞLU

Ulaş TUTAK

ANKARA

January 17, 2007

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Problem Definition

For a programmer, working with an assembly language is almost painful and by this method sophisticated projects are hard to realize. Moreover, assembly programs, being succinct and cryptic, are not very accessible. Beside these, by copying the program to the hardware for every testing is a waste of time. The development of on-chip debuggers, and onchip BASIC interpreters by an industry in which the profit margins are already slim, is a strong indicator of a need in this field.

MPLAB is very widely used by PIC developers because it is freeware and produced, distributed by the developer of the PIC MCU and thus accepted to be the most standard development tool. It also supports the widest variety of PIC families. However, it does not run on Unix-like operating systems, to which category most free operating systems belong. Beside these, it is not open source. Thus, it is not suitable for adapting to other uses and does not lend itself readily to interfacing with other applications. Moreover, trying to determine how a peripheral (e.g. an LCD) will behave in certain situations by tracing the code and watching numerical values are very cumbersome to say the least. Also, simulating more sophisticated features of PIC MCU, such as analog/digital conversion or serial communication, in MPLAB is either very hard or just totally impossible.

1.2 Purpose of the System

In DEVEMB project, we are going to develop a software development environment, compiler and emulator for a specific embedded system card which is PIC demo board that is designed for CEng 336 Embedded Systems course. There are many products in the market used for development for PIC micro-controller units in the market, compilers, debuggers and simulators. However, testing is cumbersome in that software because development environment and simulation software is not integrated. For instance, when using MPLAB IDE, one can not see the result of the execution interactively or examine the output of LCD,

whereas, when simulating a program on ISIS, basic debugging facilities such as tracing are not available to the programmer. Besides, one has to switch between several programs to get the desired result. By using DEVEMB, users will be able to compile, debug and test their programs in the virtual card emulated by software. Since, testing and simulation software is not integrated, testing is very difficult. Therefore, DEVEMB will be the software which integrates the simulation and testing. Likewise, each of the other products suffer some deficiency, be it limited capabilities, limited licenses or high prices. Being compatible with Unix OS and containing all necessary tolls for PIC 16F877 programming, DEVEMB will be a solution to those problems.

In this project we are developing software which will be used in industry and education. Therefore, our customers will be students, engineers and technicians. Our product will work correct and fast. The interface has to be easy to understand and by using our product one should develop his/her program rapidly. Also, our product should include all the necessary tools in order to program 16F877 PICmicro microcontroller.

DEVEMB is an integrated tool set for the development of embedded applications for Microchip's PIC 16F877 microcontrollers. DEVEMB runs as a 32-bit application on Windows, is easy to use and includes software components for fast application development debugging. DEVEMB serves as a software emulator, compiler and development environment for CEng Card which is used in CEng336 course. Users will be able to compile, upload and debug their programs to the card. Also they will be able to test their programs in the virtual card emulated by software.

1.3 Design Constraints

The most important design constraint is the reliability of the product. Also, our product must work fast. In this project we are developing software which will be used in industry and education. Therefore, our customers will be students, engineers and technicians. Our product will work correct and fast. As the third constraint, the interface has to be easy to understand and by using our product one should develop his/her program rapidly. Also, our product should include all the necessary tools in order to program 16F877 PICmicro microcontroller.

1.4 Features of Our Product

1.4.1 Extensibility

Although, our project is in a specific case and conditions are restricted, (as we can see from data flow diagrams) our system is designed so that it can be extended in particular case. As we will generate a basic high level compiler for example we can compile simple hex files. By extending it PIC processor can generate more complex codes. Also the emulator that we make for using DEVEMB can be similar to the PICmicro microcontroller, and the user can do all kinds of work from the screen. This will help CEng336 classes mostly.

1.4.2 Robustness

Our system will be able to manage invalid user inputs or inconsistent conditions. It provides error checking to ensure the right input format and returns errors and warnings to the user.

1.4.3 Reliability

Our system will produce the expected output for a valid input at all times. Any unexpected work or error will be reported immediately, and the system prints errors to the screen for further applications in order to protect them.

1.4.4 Functionality

The system should function according to the requirements specified in Requirements Analysis Report.

1.4.5 Usability

Our project will be user friendly. Our goal is to provide the user an easy-to-use interface. The design is chosen due to the familiarity of most users with this kind of interface. It consists of simple interface and the user who is familiar with 16F877 PICmicro microcontroller can use the system efficiently and it also provides the user more simple features to understand. The

project provides user/system interaction. The user is placed in a familiar environment, which eases the general use of the application.

1.4.6 Conciseness

When we start to the implementation of our project, we will give importance to produce concise code. This will help us in two different ways in the future:

- The resulting code will be more maintainable.
- The resulting executable will be small.

1.4.7 Efficiency

We forego system independence in favour of performance. We also forego some encapsulation to make the simulator as fast as possible. Moreover, in order to provide efficiency:

- Our program will be deployed in native code instead of interpreted intermediate code.
- The access to simulator variables will not be through accessory methods.

1.4.8 Portability

Our understanding of portability is different from the general view. Instead of compiling the same code on all systems with no change, we will produce a program that runs on windows but it does not require you to

- have right to install programs,
- clean littered files and directories from your computer,
- have hundreds of megabytes of space on your hard drive,
- have the fastest CPU or lots of RAM.

Basically, if we reach our design goals, you will be able to copy the program directory to a flash disk and go to any lab or an internet cafe to do your CENG336 homework.

1.5 Overview

Final Design Report explains the design of our application in detail and provides detailed information about functionality and implementation of our project. This report is extended and revised version of the Initial Design Report. For this report, we have taken some unnecessary parts that exist in the Initial Design Report out, we have changed some parts and we have added some extra topics. Also, we emphasized the special features of our product, and explained our solution in detail with the help of the numerous diagrams. Basically, throughout this document the following subsections will be stated:

Problem definition, design constraints, features of our product, our approach to the problem, file formats and tools that we are planning to use for the implementation.

While explaining our approach we mentioned our modules. To tell our product and its modules, we draw dynamic and static diagrams. In the UML Diagrams sections, the reader can find Use-Cases, Class Diagrams, Class Descriptions and Sequential Diagrams. Some of these diagrams are explained more briefly with descriptions. Additionally, we add explanations about all our classes and their methods and attributes. Finally, we mentioned file formats and tools that we are planning to use for the implementation.

This report aims to provide facility to any programmer to implement this project that reads this report and follows the diagrams and explanations.

2. BEHAVIORAL DESCRIPTION OF THE DESIGN

2.1 Simulation Module

2.1.1 Simulation

2.1.1.1 Use-Case: Simulation

Simulate Program

User can load a program to simulator, configure the simulator, run or debug the program and interact with the simulator.

Preconditions: There must be a compiled program or a binary file.

Trigger: User start the simulator by clicking a button from the user interface.

Basic Course of Events: User should start the simulator. User opens a file. User configures the simulator. User starts the simulation. Finally, either the program ends normally or user halts it.

Post Conditions: Output files must be created. Output files must be saved and closed.

Configure Simulator

User can configure the clock, and stimuli.

Preconditions: As explained in simulate program, user must entered to simulation mode but simulation must not be started.

Trigger: User can click load stimuli button or changes the clocking values from the interface.

Basic Course of Events: User should select configure mode. Configuration pop-up appears. User changes the settings or values that s/he wants. User can click the OK or CANCEL buttons to finish the configuration mode.

Post Conditions: Changes must be saved to a configuration file.

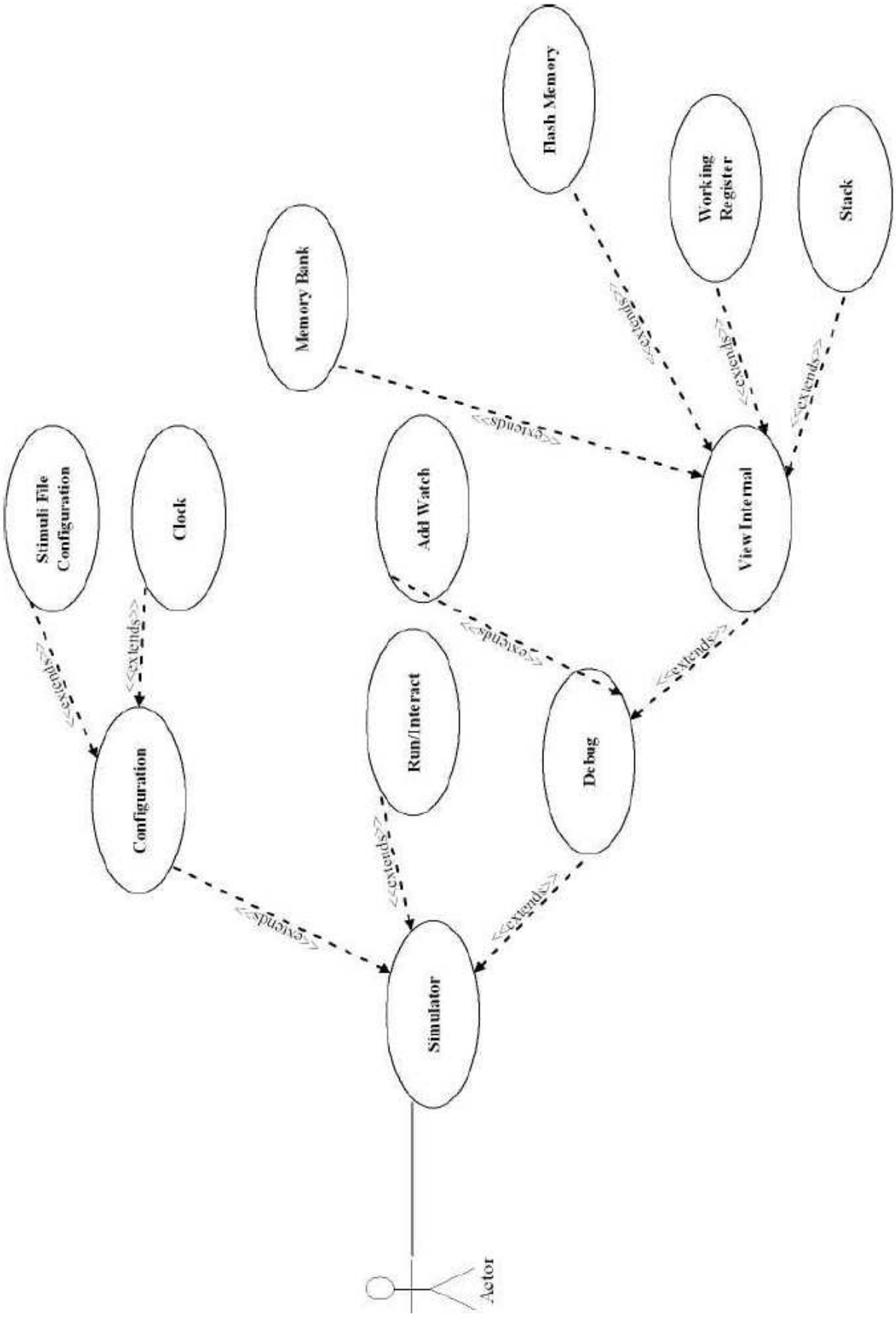


Figure 2.1: Use-Case: Simulation

Run Simulation

This use case shows the interaction of user with the simulator after starting the simulation.

Preconditions: Simulation mode

Trigger: User interface

Basic Course of Events: User starts the simulation. After starting simulation user can click buttons. Simulation ends normally or user halts it. Simulator shows the changes on the LEDs, LCD, etc.

Post Condition: Stimulus files must be created in this phase.

2.1.1.2 Sequence Diagrams

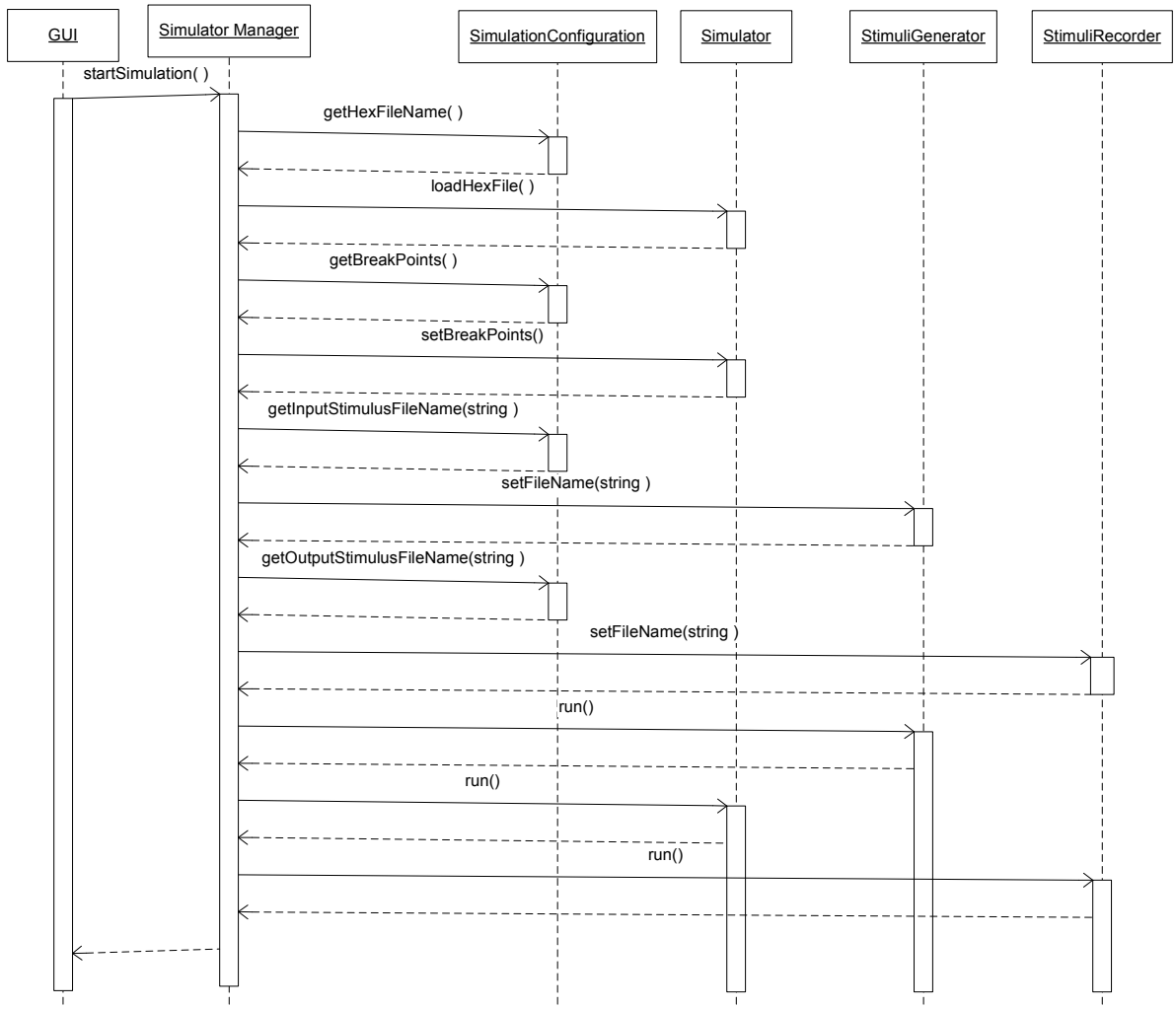


Figure 2.2: Simulation Setup

Simulation Cycle 1

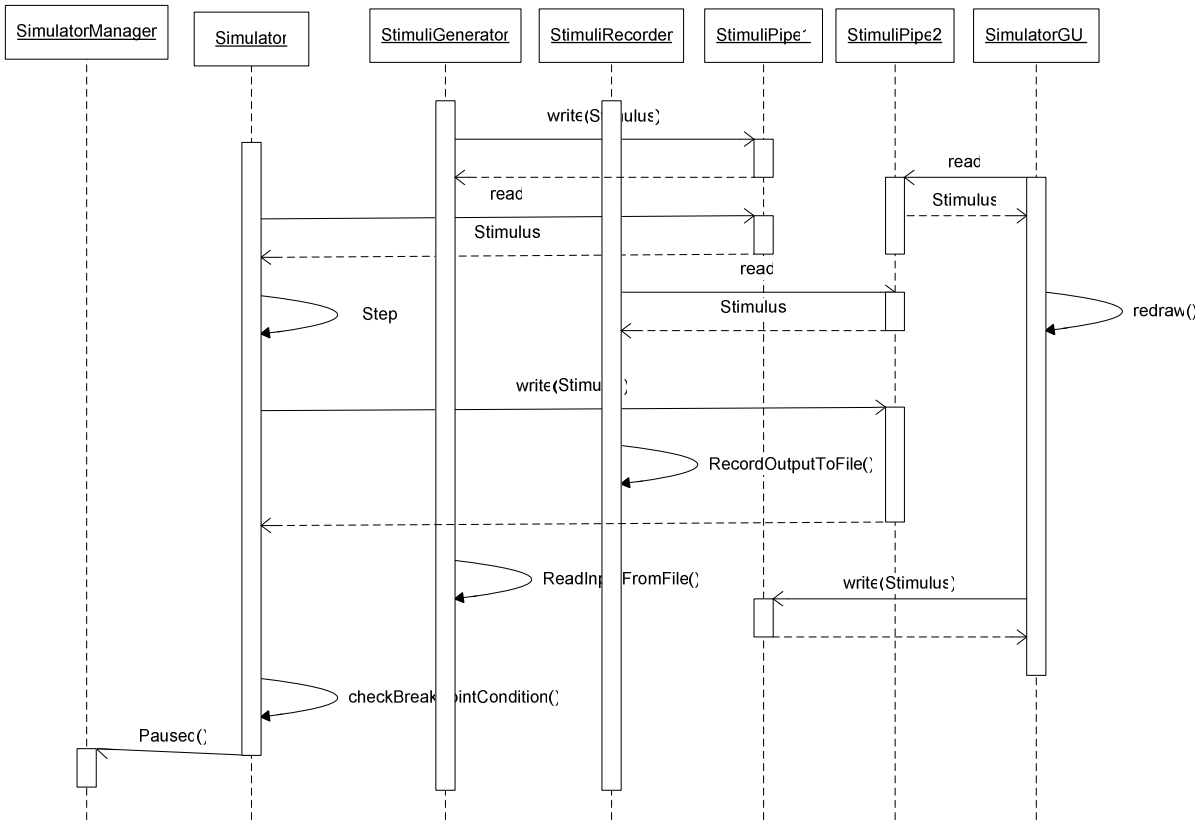


Figure 2.3: Simulation Cycle 1

Simulation Cycle 2

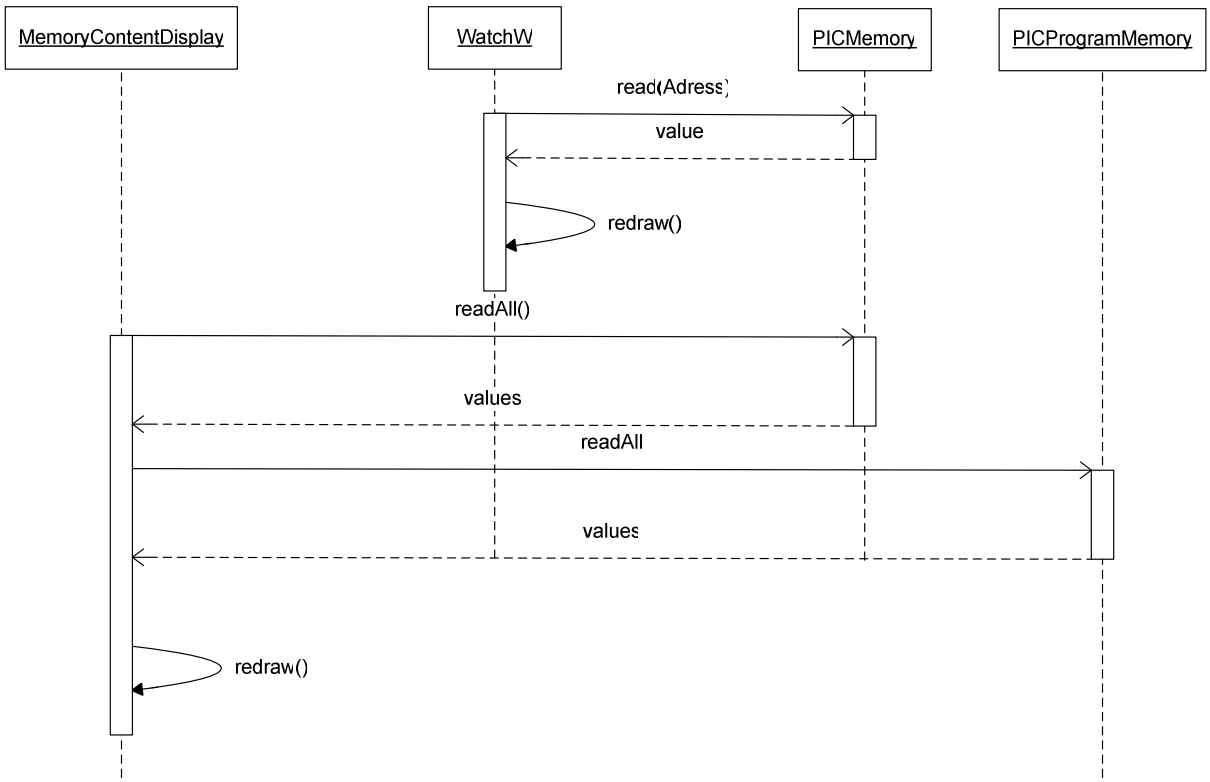


Figure 2.4: Simulation Cycle 2

Simulation Breakpoint Reached

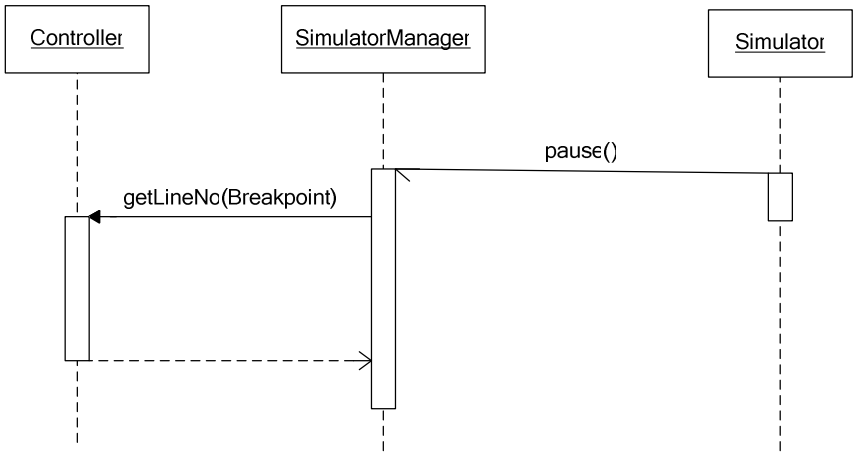


Figure 2.5: Simulation Breakpoint Reached

Simulation Pause

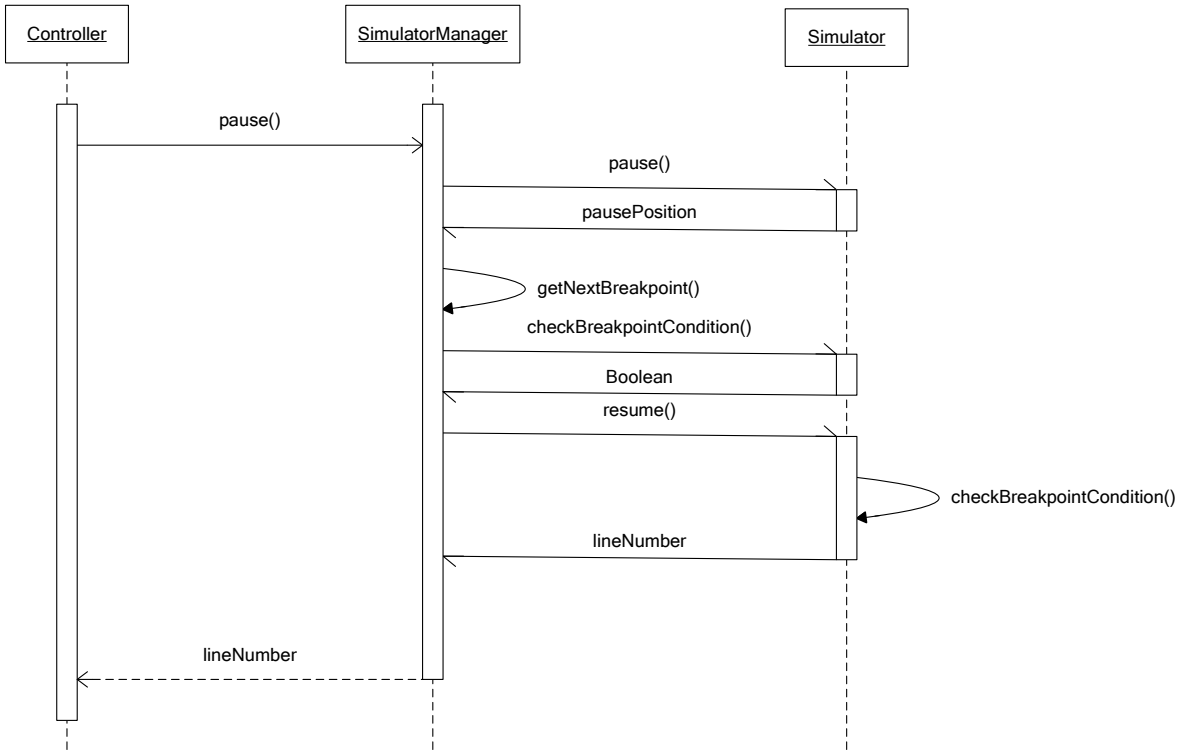


Figure 2.6: Simulation Pause

Simulation Halt

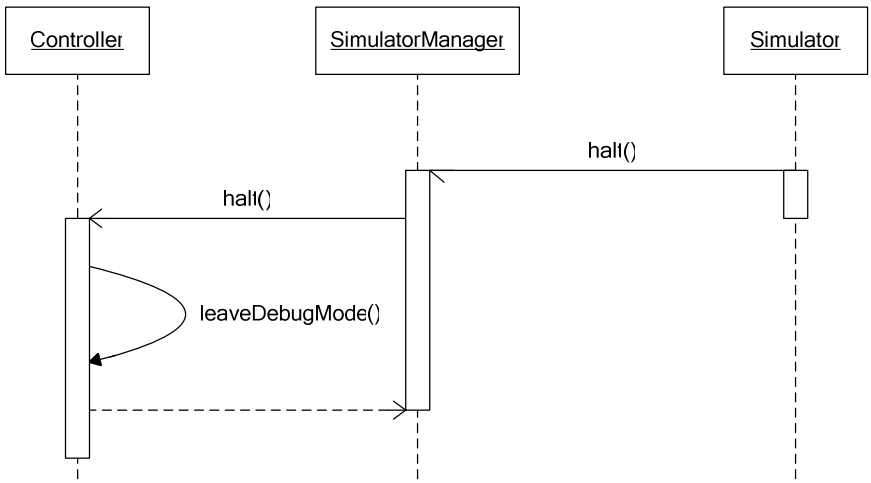


Figure 2.8: Simulation Halt

Simulation Step

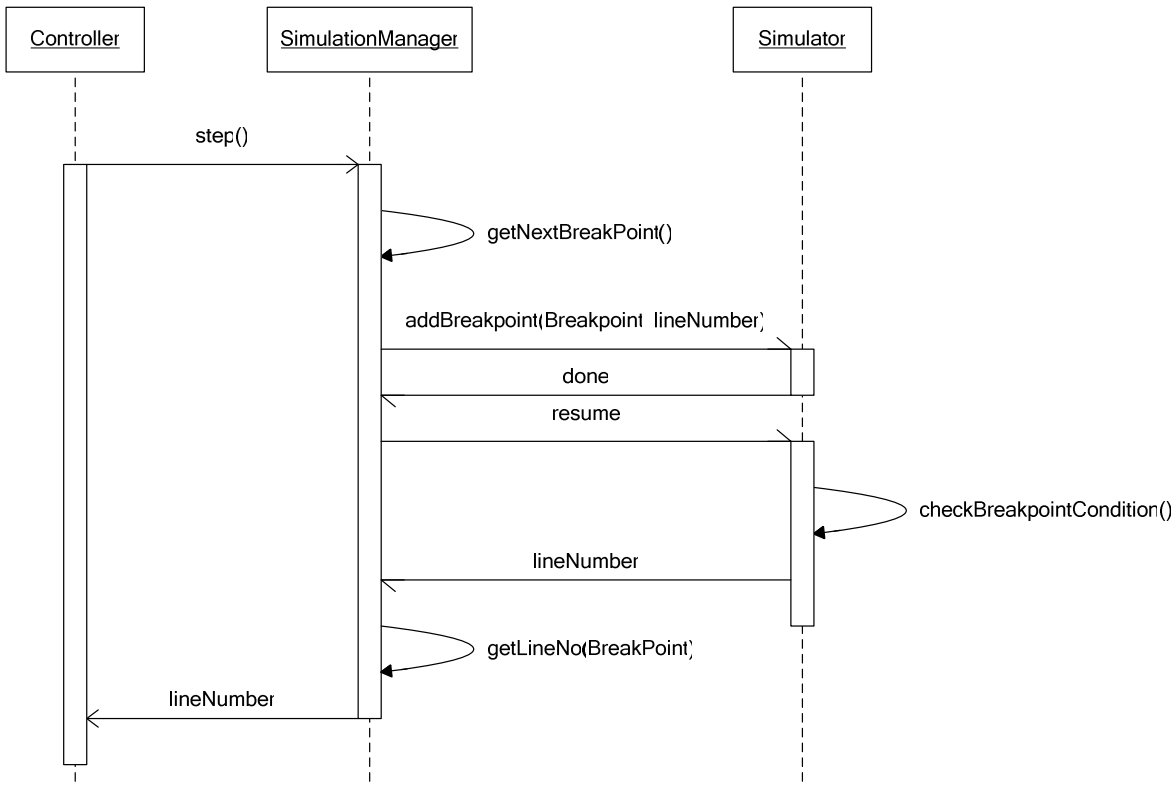


Figure 2.7: Simulation Step

2.1.1.3 Activity Diagram

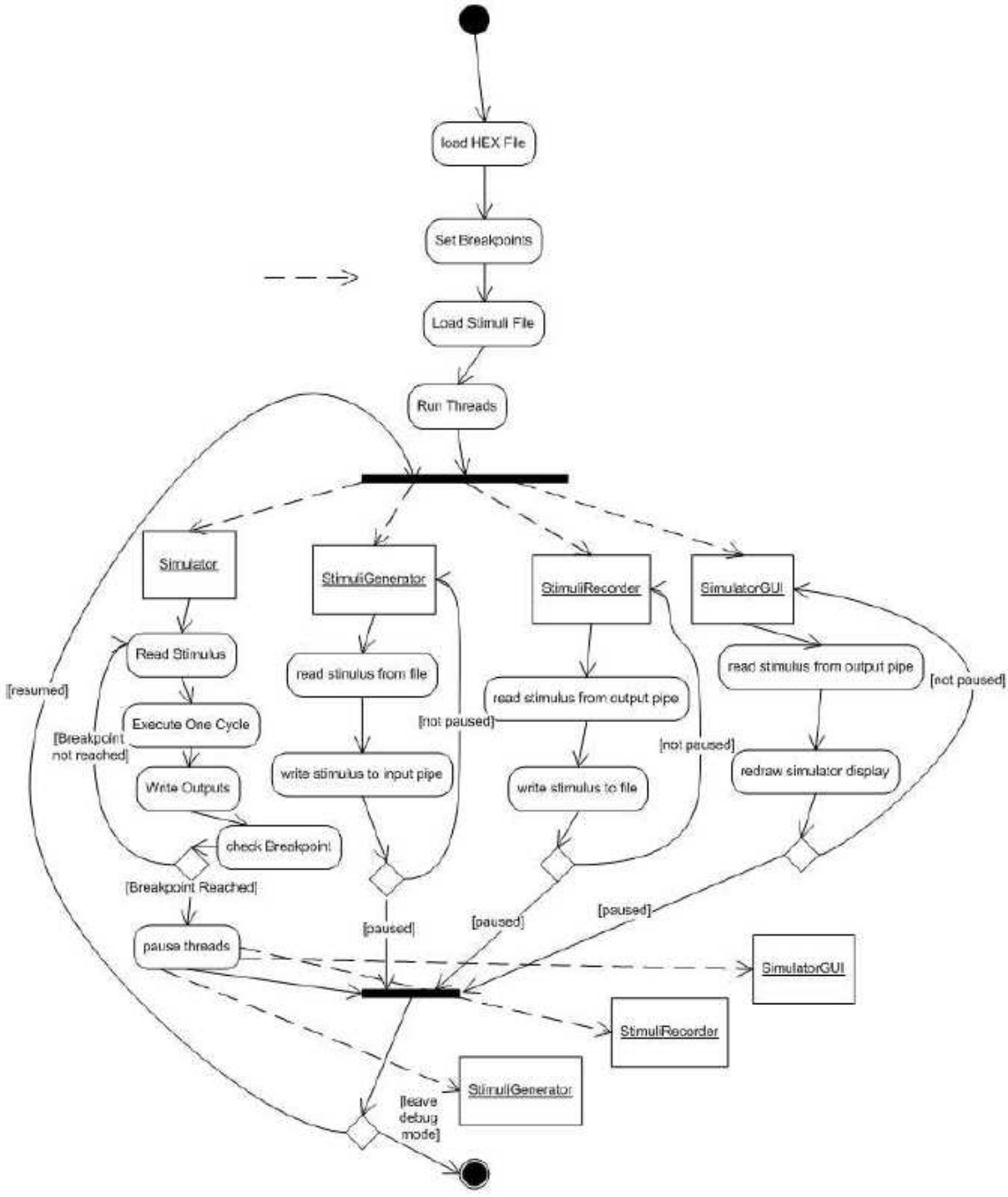


Figure 2.9: Simulator Activity Diagram

2.1.2 Debug

2.1.2.1 Use-Case: Debugging the Code

User can trace the program view contents of the memory.

Preconditions: There must be a program which is compiled for debugging or binary file.

Trigger: User start the debugger by clicking a button from the user interface.

Basic Course of Events: User selects the memory location that s/he wants to see the content. User puts the breakpoints if there is. User traces the program. Debugger shows the contents of the selected memory locations. Debugger ends normally or user halts it.

2.1.2.2 Sequence Diagrams

Debug Add Watch

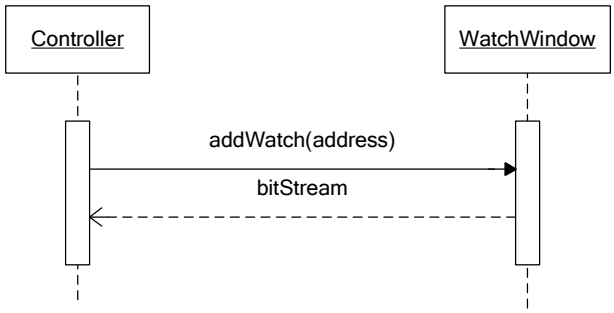


Figure 2.10: Debug Add Watch

Debug Delete Watch

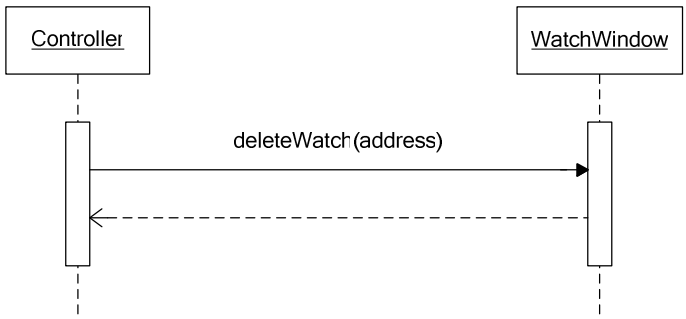


Figure 2.11: Debug Delete Watch

2.1.2.3 Activity Diagram

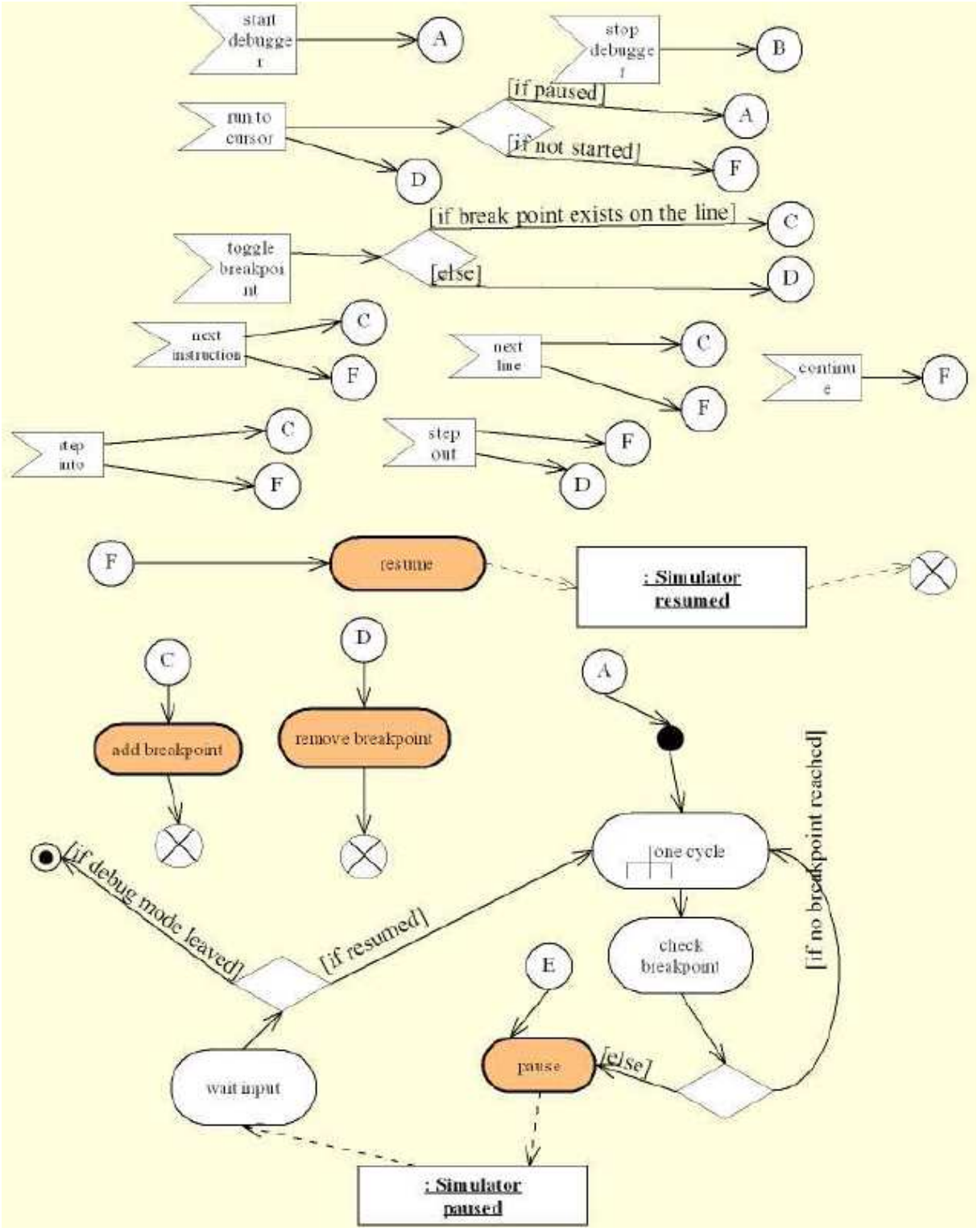


Figure 2.12: Debug Mode Activity Diagram

2.2 Compilation Module

2.2.1 Use-Case: Compilation

User may chose to generate pic machine codes and debugging information from C and Assembly source files opened in the editor component of the user interface or generate assembly code from pic machine instructions.

Compile a C Program

User can compile a program to generate output file(s) in a specified format.

Preconditions: There must be an existing C source file.

Trigger: User starts the compiler by clicking a button or selecting a menu item labelled appropriately from the user interface.

Basic Course of Events: User starts the compilation. Compiler program processes the source file. The compiler might generate warnings. It either generates error messages and quits without completion of the compilation or completes the compilation and reports success. Any messages generated by the compiler are listed in a window.

Post Condition: Output files, e.g. replaceable object code (.o), assembler source (.asm), assembler listing (.lst), executable for debugging/simulation file (.cod), executable for programming the processor (.hex), depending on the compilation options and the level of the compilation reached when the compiler program halts.

Assemble an Assembly Program

User can assemble an .asm file into one or more of the output formats mentioned above.

Preconditions: There must be an existing assembly source file.

Trigger: User starts the assembler by clicking a button or selecting a menu item labelled appropriately from the user interface.

Basic Course of Events: Basic course of events are the same as compilation use-case.

Post Conditions: Post conditions are the same as compilation use-case.

Disassemble an Executable

User can assemble a .hex file into a .asm file.

Preconditions: There must be an existing .hex file.

Trigger: User starts the dis-assembler by clicking a button or selecting a menu item labelled appropriately from the user interface and selecting an executable hex file from a file dialog. Unlike compile and assemble use-cases in dis-assemble use-case the input file need not be open at the time of processing, as opening a hex file in an editor makes little sense.

Basic Course of Events: User starts the dis-assembly. Dis-assembler program processes the hex file. It either generates error messages (e.g. if invalid op-codes are present in the input file) and quits without completion of the dis-assembly or completes the dis-assembly and reports success. Any messages generated by the dis-assembler are listed in a window.

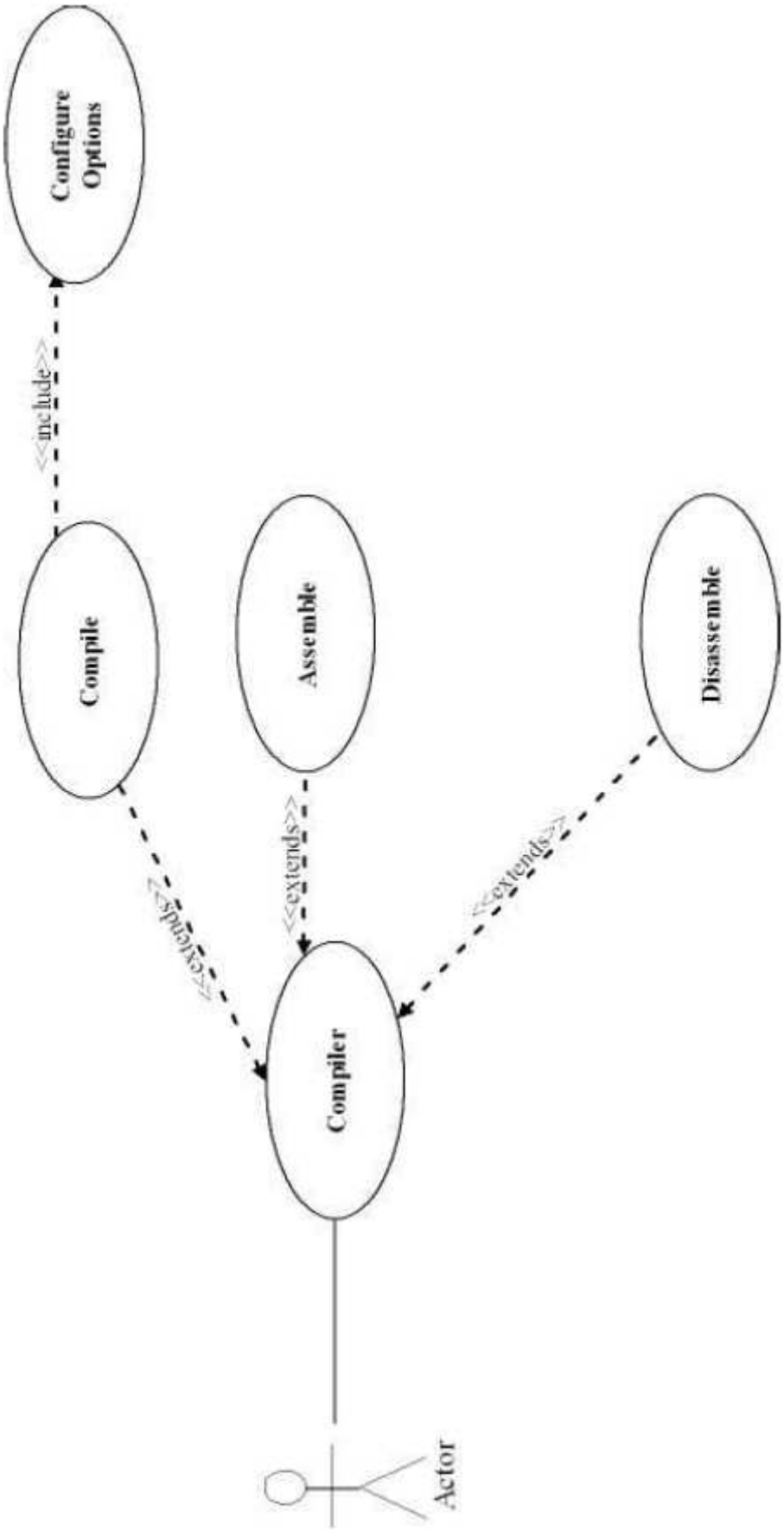


Figure 2.13: Use-Case: Compilation

2.3 Program Loading Module

2.3.1 Use-Case: Load Program

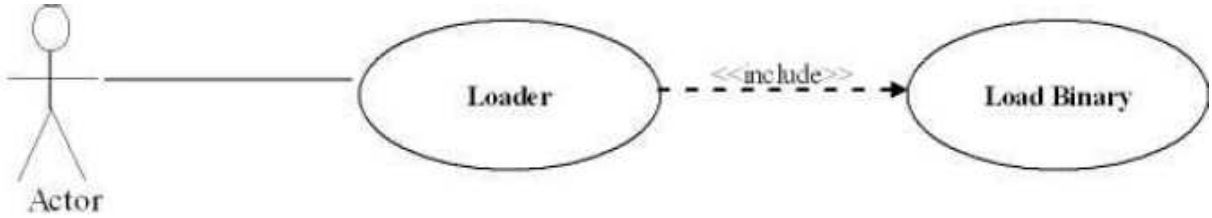


Figure 2.14: Use-Case: Load Program

User can assemble a .hex file into an .asm file.

Preconditions: There must be an existing .hex file.

Trigger: User starts the dis-assembler by clicking a button or selecting a menu item labelled appropriately from the user interface and selecting an executable hex file from a file dialog. Unlike compile and assemble use-cases in dis-assemble use-case the input file need not be open at the time of processing, as opening a hex file in an editor makes little sense.

Basic Course of Events: User starts the dis-assembly. Dis-assembler program processes the hex file. It either generates error messages (e.g. if invalid op-codes are present in the input file) and quits without completion of the dis-assembly or completes the dis-assembly and reports success. Any messages generated by the dis-assembler are listed in a window.

2.3.2 Sequence Diagrams

Load Executable to Board

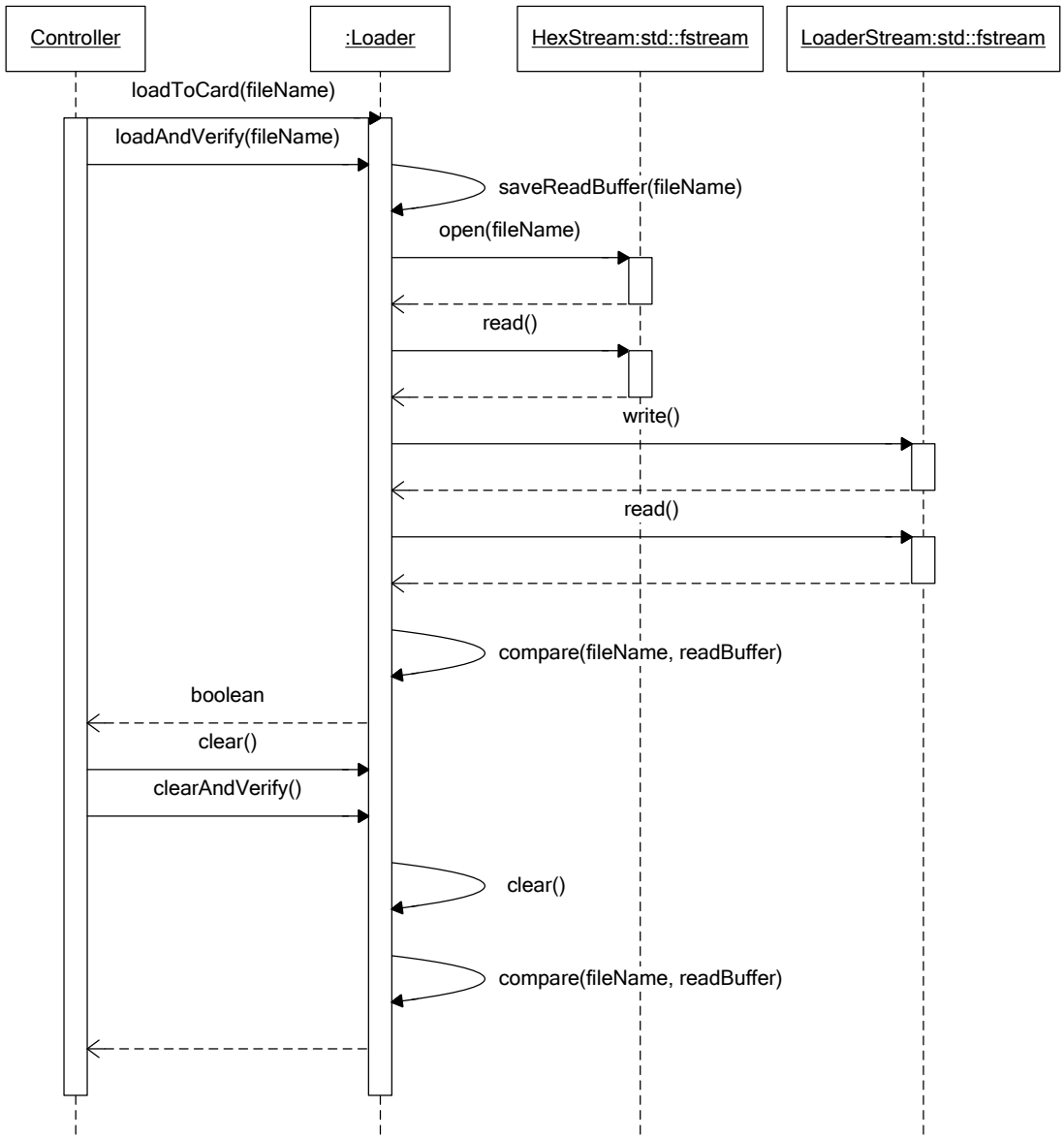


Figure 2.15: Load Executable to Board

2.4 Project Management Module

2.4.1 Use-Case: Project Manager

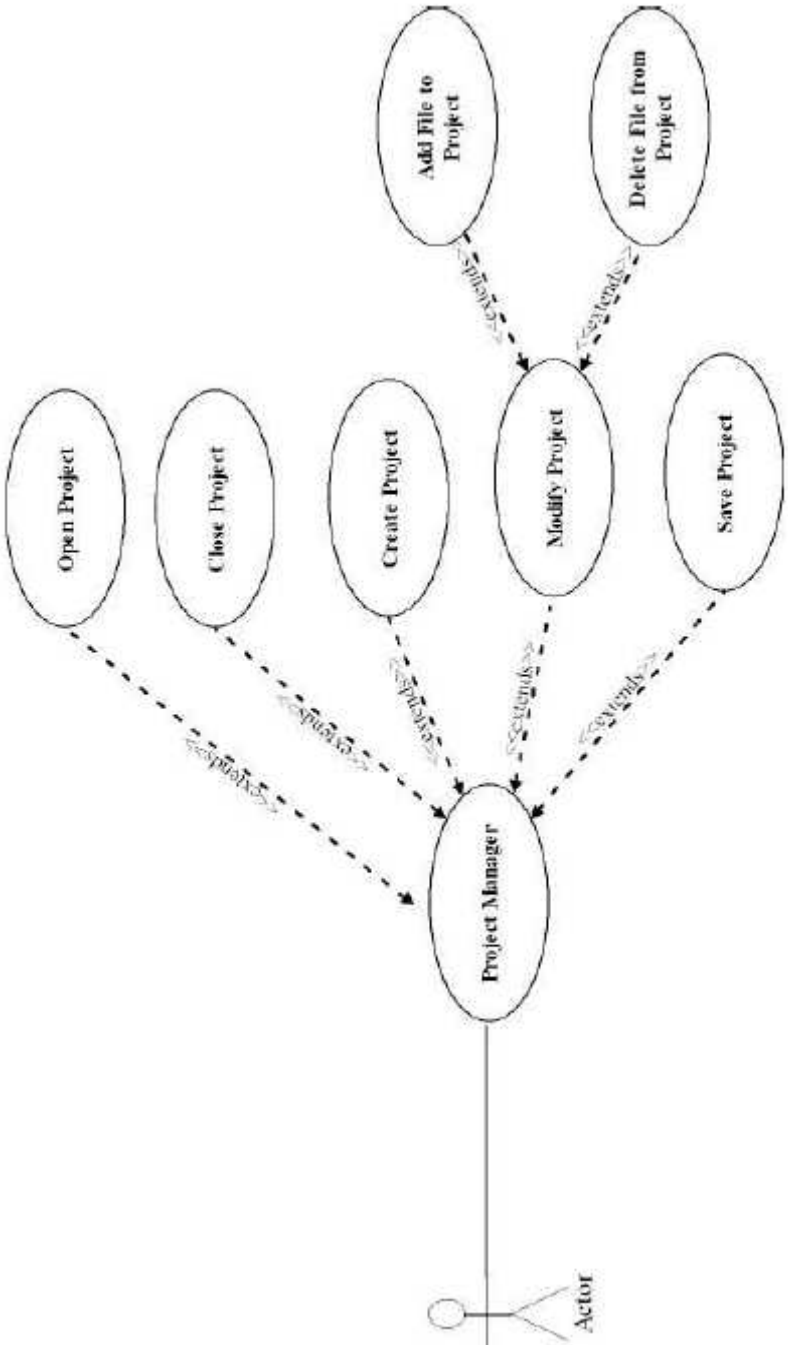


Figure 2.16: Use-Case: Project Manager

Preconditions: DEVEMB must be executed.

Trigger: User starts managing the project by clicking a button from the user interface and also can see the files of the project from the user menu.

Basic Course of Events: User should open the project. User should close the project. User should create the project. User should modify the project. User should save the project.

Post Condition: Project should be closed.

Modifying a Project

Preconditions: A project must be opened.

Trigger: User starts modifying the project from the user menu.

Basic Course of Events: User should add a file to the project. User should delete a file to the project.

Post Condition: New configuration must be set.

2.4.2 Sequence Diagrams

Add a File to Project

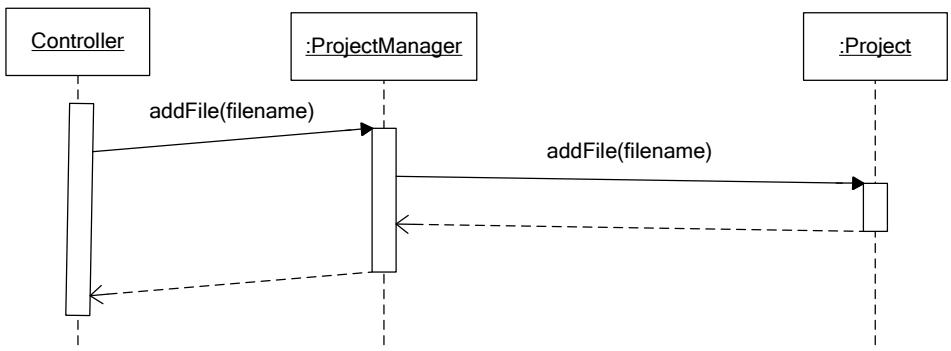


Figure 2.17: Add a File to Project

Remove a File from Project

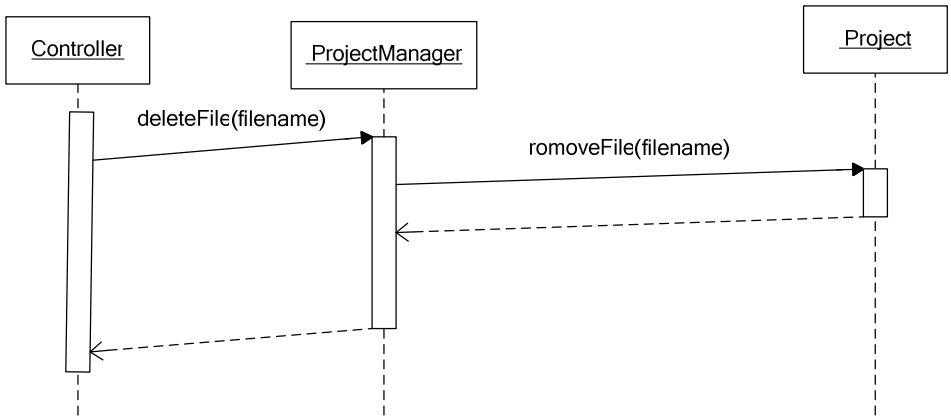


Figure 2.18: Remove a File from Project

Close Project

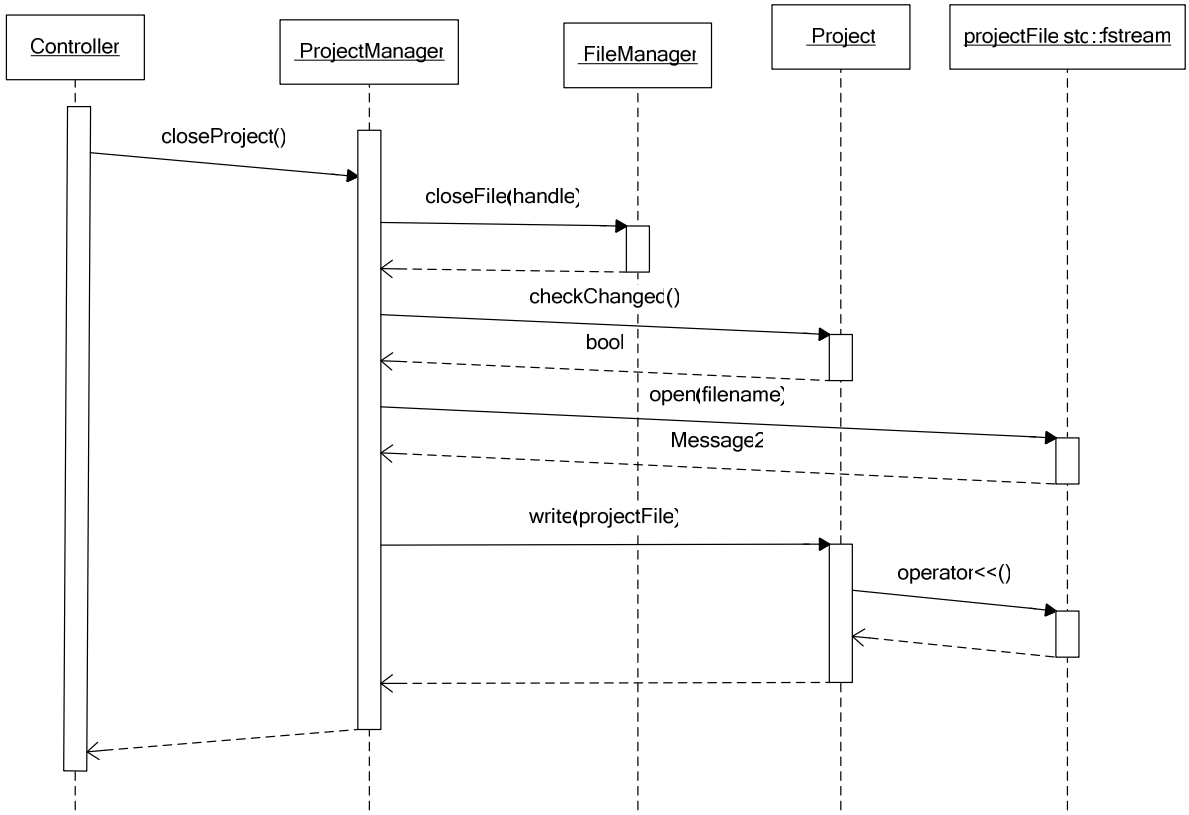


Figure 2.19: Close Project

Open Project

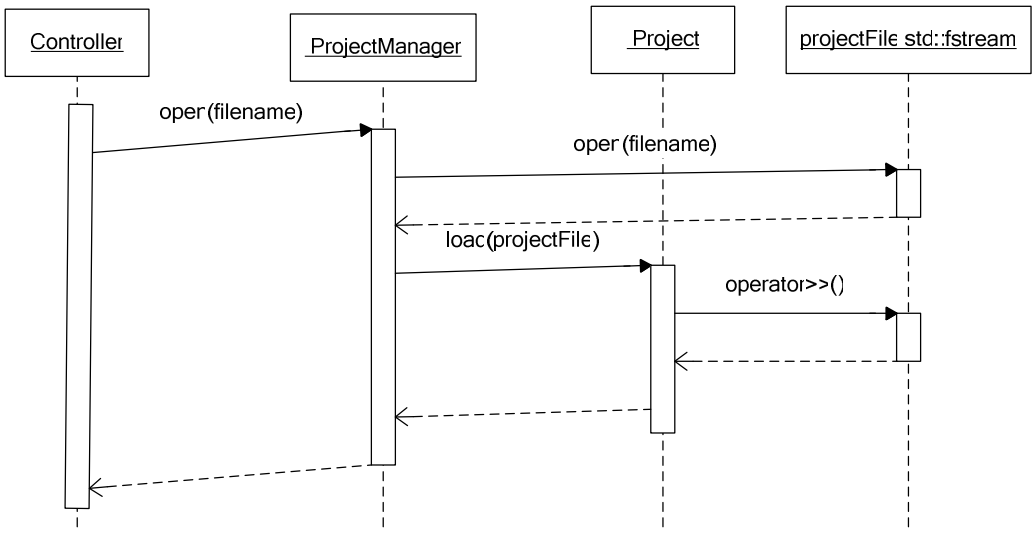


Figure 2.20: Open Project

Save Project

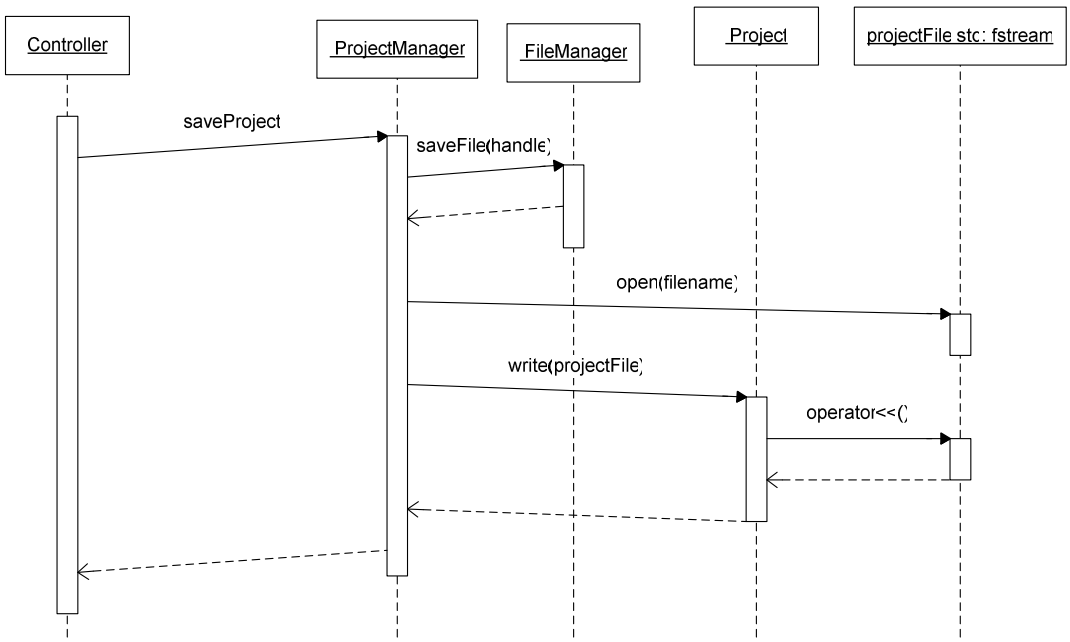


Figure 2.21: Save Project

2.5 File Management Module

2.5.1 Use-Case: File Manager

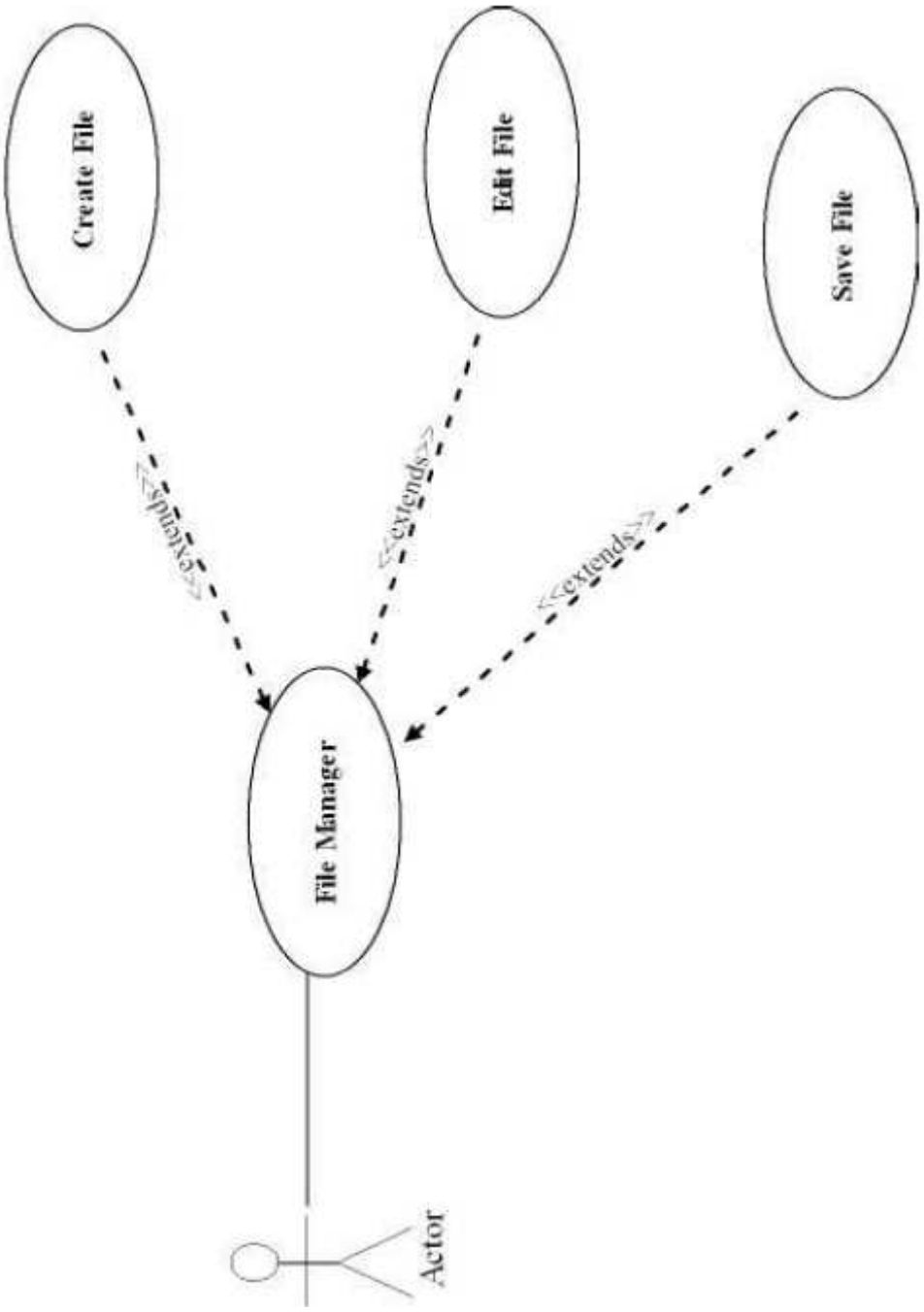


Figure 2.22: Use-Case: File Manager

Preconditions DEVEMB must be executed.

Trigger: User start managing the files by clicking a button from the user interface.

Basic Course of Events: User should edit a file, create a file, close a file or save a file.

Post Condition: Files should be closed.

2.5.2 Sequence Diagrams

Close File

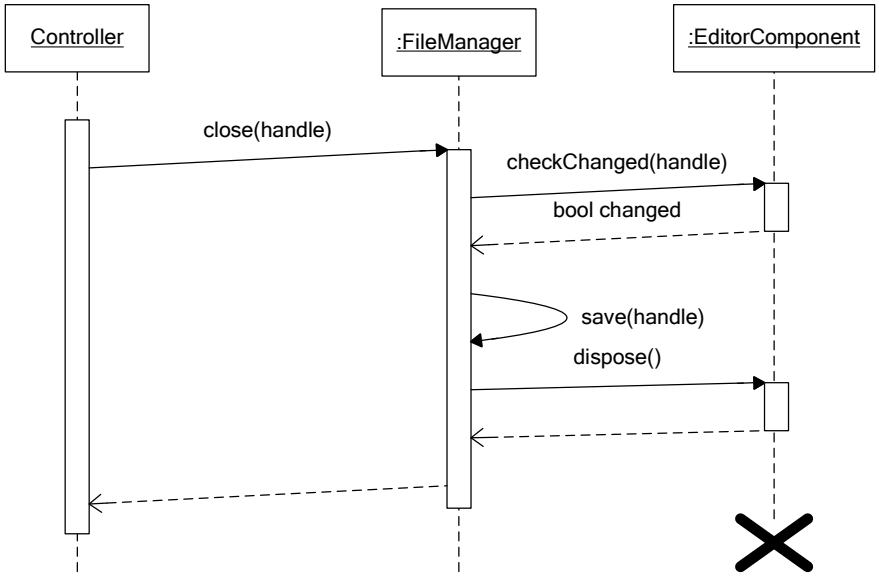


Figure 2.23: Close File

New File

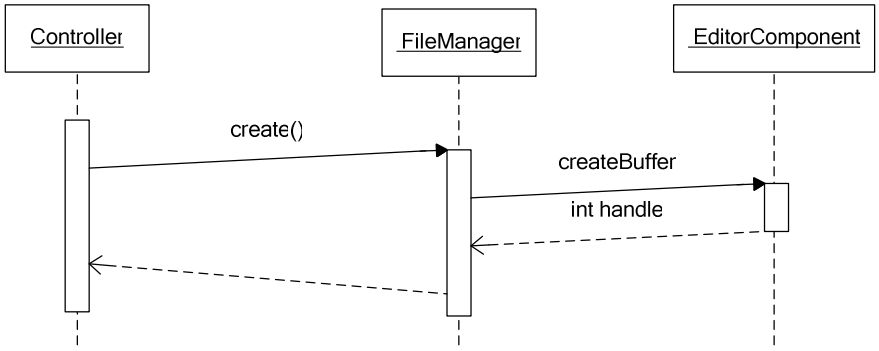


Figure 2.24: New File

Open File

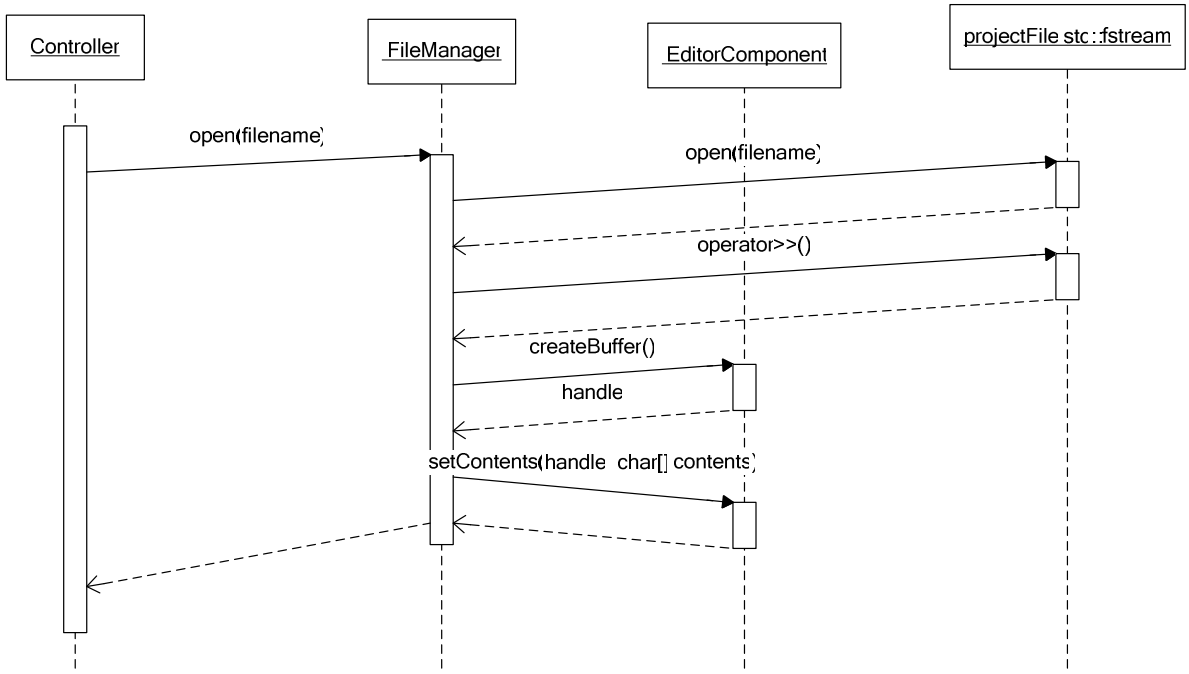


Figure 2.25: Open File

Save File

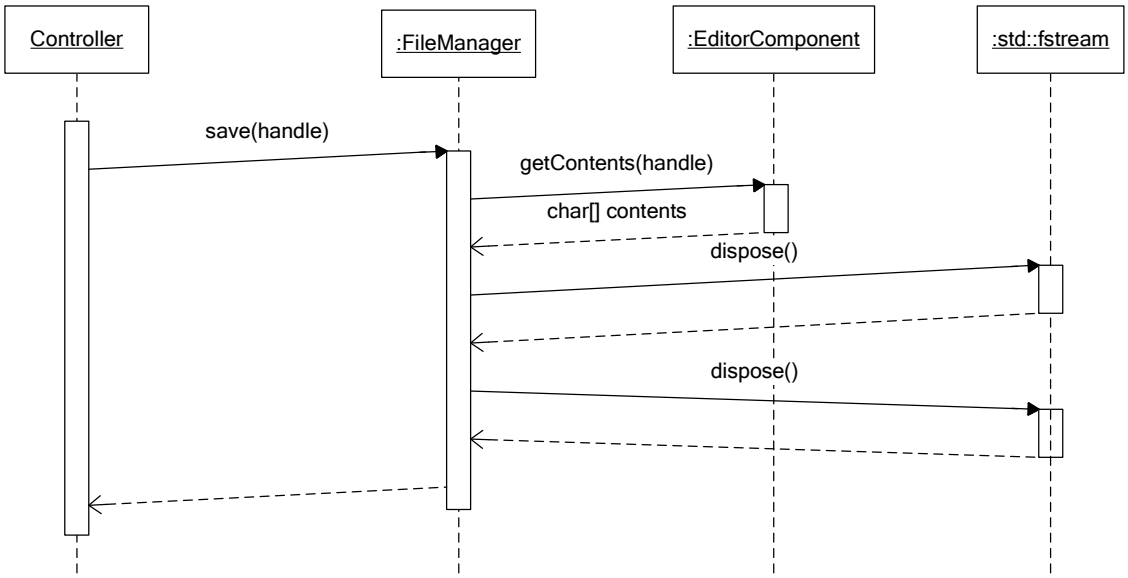


Figure 2.26: Save File

3. STATIC STRUCTURE

3.1. WatchWindow Class

Viewing the contents of the specified memory addresses is very important issue for the PIC programmers according to our CENG 336 course experiences because testing or tracing the program can be realized by viewing the specified or important registers contents. In order to facilitate viewing the contents of the specified memory addresses, we decided to add a WatchWindow class. WatchWindow class has an attribute which is addresses. This class has also three functions named as WatchWindow(), addWatch(int address) and deleteWatch(int address).

addresses: addresses property is a collection of individual addresses. Each individual address is an entry in this collection.

WatchWindow(): This function is the constructor of the WatchWindow class. It creates an empty vector named as addresses.

addWatch(int address): This function gets an argument which is address and it returns a bit stream whose length is 8. addWatch (int address) adds an entry into the addresses collection.

addWatch (int address) function gets the content of the register specified by the address. After that it returns the content as an 8 bit stream.

deleteWatch(int address): This function gets an argument which is address that shows which address will be deleted. deleteWatch (int address) function gets the integer value then looks to the addresses. If there is an address there, it will be deleted. If the given address does not exist, it returns silently without doing anything further.

3.2. Loader Class

Programmers always do not write program and test them. Sometimes they may want to examine or test any program written by any other programmers. Therefore, our product has to facilitate to load any program file to the PIC, execute and test it. For this purpose we think that we will need a class to implement this load facility. Therefore, we have added Loader class to our design. loader class has two buffers to hold binary program data: Read buffer and write buffer. Moreover, loader class has seven functions. These functions are saveReadBuffer(const char* filename), loadFromCard(), loadToCard(const char* filename), compare(const char* filename, byte* readBuffer), clear(), loadAndVerify(const char* filename) and clearAndVerify() functions.

saveReadBuffer(const char* filename): This function gets an argument which is filename that shows the name of the file where we will save the read buffer. This method will create a file that is named as filename then open it and save the content of the read buffer to this created file.

Bool loadFromCard(): This function gets no argument and returns Boolean. loadFromCard() reads the hexfile from the PIC card loaded before and writes this hexfile to the read buffer. If any error does not occur while reading from PIC card and writing to read buffer, this method returns true. However, if there is any error while read and write process, it returns false.

Bool loadToCard(const char* filename): If the loading process starts in a normal way, it will return true. Else if loading process does not start, for example we do not have write permission to the Parallel port of the computer or the computer does not have a parallel port at all, or it gives error while loading, this function will return false. It loads the hex file to the write buffer, using HexFileCoDec class. It opens the parallel port using system specific API and write the contents of the hexfile according to the protocol used in the programming hardware of the PDB (propic2 programmer).

Bool compare(const char* filename, byte* readBuffer): This function gets two arguments. One of these is the filename that shows the name of the file which we will compare with read buffer and the other argument is readBuffer, this shows the hexfile loaded before to the PIC board. This program starts to compare these two file from start to end. If there is no difference between these two file, the program will return true. This will show to the user the program that is loaded is ready to execute. Else if there is any difference between these two files, it will return false. This result shows to the user while loading the program, some errors occurred. Therefore, program could not be loaded correctly. Thus, to execute the program, it has to be loaded again.

clear(): This function gets no arguments. clear() method deletes the hexfile loaded before to the PIC board.

Bool loadAndVerify(const char* filename): This function gets an argument which is filename. In order to perform its duty loadAndVerify(const char* filename) method calls firstly the loadToCard(const char* filename) method and then it calls the compare(const char* filename, byte* readBuffer) method. As a result, it returns what it gets from these two functions. Thus, if the hexfile is loaded to PIC card without any problem and the comparison between the read buffer and the file is successful, it will return true. Else if there is any problem while loading or comparison, it returns false.

clearAndVerify(): This function gets no argument. While performing its function, it calls firstly the clear() method and then it calls the compare(const char* filename, byte* readBuffer) method.

3.3. Simulator Class

Simulator class represents the 16f877A MCU. After writing the program, programmer may want to execute and see the results of the program or if there is any erroneous situation in his/her program programmer may want to debug the program to find the erroneous case. Therefore, we have to offer the user, simulation option to satisfy these demands. Our product will have the simulator facility and to realize this purpose a simulator class is necessary for our implementation. Simulator class has two attributes which are address and pausePosition. Our simulator class has ten functions. These functions are: start(), halt(), pause(), resume(), removeBreakpoint(int lineNumber), addWatch(int address), adjustClockFrequency(), step(), checkBreakpointCondition(), and addBreakpoint(Breakpoint breakPoint, int lineNumber).

addresses: Addresses property is a collection of individual addresses. Each individual address is an entry in this collection.

pausePosition: This is a global variable and its type is int.

start(): This function gets no argument and it starts the simulator according to specified setting which are determined by the user using some other methods such as addWatch(int address), adjustClockFrequency(), addBreakpoint(Breakpoint breakpoint, int lineNumber), etc.

halt(): This method ends the simulation abnormally. If the user wants to finish simulation of any program without waiting end of the program, s/he clicks the End Simulation button from the user interface. Then, our program executes this halt() method and ends the simulation mode.

int pause(): This function is used to stop the simulator for a while. If the user wants to stop the simulation process for some time, s/he clicks the Pause button by the user interface. After that, our program will calls this pause() method and pause() method will stop the simulation process for a while and it returns the line number at which it is called. This line number is assigned to a global variable whose name is pausePosition to use later. This global variable which is assigned the line number can be used later to resume the simulation process if it is desired.

resume(int pausePosition): This function gets the global pausePosition variable as an argument. resume(int pausePosition) method will be used to start the simulation process again after it paused. When the simulator is paused, the line number of the pause operation is saved to the pausePosition. Therefore, we will know where the simulator is paused by this way and we will use pausePosition variable to decide to the where or at which line we should start the simulation process again by using resume(int pausePosition) method.

removeBreakpoint(int lineNumber): This method gets an integer whose name is lineNumber. When this function is called, if any breakpoint has been set before for given that lineNumber given as an argument, removeBreakpoint(int lineNumber) function calls the destructor of the Breakpoint class for given that lineNumber. Else if any breakpoint has not been set before for that lineNumber, it returns without doing anything.

addWatch(int address): This function gets an argument which is address and it returns a bit stream whose length is 8. addWatch (int address) adds an entry into the address collection.

addWatch (int address) function gets the content of the register specified by the address. After that it returns the content as an 8 bit stream.

adjustClockFrequency(int frequency): This method is used to adjust the clock frequency of the PIC according to the argument whose name is frequency.

step(): This method fetches and executes the next instruction. The state of the MCU, hold in the registers are updated accordingly. Counters, e.g. those used by interrupt timers, are incremented.

Bool checkBreakpointCondition(): This function gets no argument and checks whether there is any breakpoint on the current line. If there is any breakpoint set before on the current line, it returns true. Else if there is not any breakpoint on the current line, it will return false.

addBreakpoint(Breakpoint breakPoint, int lineNumber): This function gets an argument which is a Breakpoint. When this method is called, it calls the constructor of the Breakpoint class for the given lineNumber.

3.4. StimuliPipe Class

After writing code, to see the results of written program or to trace the program that we write we need to simulate it. Therefore, to perform the simulation process, we need some records to display on the screen and we need to time intervals to understand which record should be displayed when on the screen. StimuliGenerator produces some records to perform the simulation process. Some stimuli such as those coming from or going to the USART component, requires us to process each of them, and process them in order. For example when we write “Hello world!” to the serial interface, since the frequency of simulator, baud rate of the port, our polling of the stimuli source e.g. the keyboard, a virtual console, a file etc. our update rate of the outgoing stimuli, e.g. the refresh rate of the display component or console, our writing speed to the file, cannot be synchronized. These IO operations must be buffered. Therefore, we have added this StimuliPipe classes to our project. StimuliPipe1 and StimuliPipe2 are instances of the StimuliPipe class. StimuliPipe classes have two functions. These two functions’ names are write() and read().

write(): This function gets two arguments: One of them is a string that shows the filename to which we to record or save our Stimulus. The other argument is Stimulus that should be recorded or saved to specified file. If this function finds the specified file and saves the record that is given as an argument to this file, it terminates normally. However, it could not file

name that is given as an argument or it could not save the Stimulus record to the given file, this function returns abnormally.

read(): This function gets no argument. The purpose of this function is to read the Stimulus records generated by StimuliGenerator. If this read function reads the Stimulus record from StimuliGenerator, it terminates normally. Else if this function could not read the Stimulus record fully from the StimuliGenerator, it gives an error and returns abnormally.

3.5. ProjectWindow Class

This class will be used to show the project properties and its components such as its methods' names. This project window is used for accessing the methods of the current project easily and rapidly. ProjectWindow class has an attribute which is methodNames. Moreover, ProjectWindow class has two methods named as ProjectWindow() and display().

methodNames: methodNames property is a collection of individual method names of the current project. Each individual method name is an entry in this collection.

ProjectWindow(): This function is the constructor of the ProjectWindow class. It creates an empty vector named as methodNames.

display(): This method shows the methods' names of the current project. When this project is called, it gets the stored names of the methods in order from the methodNames vector and displays them in the project window part of the user interface.

3.6. Thread Class

This class provides a wrapper for a subset of sytem API's thread related
functionality. it is intended to be more user friendly than the win32 API
code (and somewhat similar to Java API's).

attributes:

CriticalSection cs;
this is a private attribute of the class that
helps synchronization within thread.

methods:

init()

run()

pause()

resume()

halt()

enterCriticalSection()

leaveCriticalSection()

LPVOID threadEntryPoint(LPVOID lpParam): This function a system(Win32) dependent part of the class. It calls the init and run functions and thread disposal related code.

3.7. CompilerWrapper Class

This class provides a wrapper for shell commands that will be run in the course of compilation related actions.

attributes:

char *commandBuffer;

char *responseBuffer;

methods:

Each method runs a system command -generally name of a separate executable with associated command-line parameters- after putting it in the commandBuffer, and collects of the output in the responseBuffer.

compile(string& filename)

compileC(string& filename)

assemble(string& filename)

link(vector<string> filenames)

build(string& file)

buildProject(string& file)

buildSingleFile(string& file)

3.8. Breakpoint Class

constructor **Breakpoint(int instAddr):** This creates a Breakpoint object.

constructor **Breakpoint(string source, int lineNo):** This creates a Breakpoint object at the address that corresponds to the given statement in the source file at the given line no.

3.9. HexFileCoDec Class

upload(const char* filename, byte* buffer): Copies the binary info encoded in the given hex file into the buffer.
download(const char* filename, byte* buffer): Copies the binary info from the buffer into a file created with the given file name encoded in HEX file format.

3.10. PICMemory Class

attributes:
mem byte[MEM_SIZE]
progmem byte[PROG_MEM_SIZE]

methods:
inline void write (addr_t addr, byte val): write a byte of data to the memory.
inline byte read (addr_t addr): read a byte of data from the memory.
void loadProgram(const char* filename): This function copies the contents of the given hex file to program memory. It uses HexFileCodec class to that effect.

3.11. SimulationConfiguration Class

SimulationConfiguration class encapsulates the information necessary to be set prior to the start of a simulation session. This information consists of the clocking, input and output port redirections, and the program to be run. The properties of the SimulationConfiguration therefore are frequency, throttle, inputfilenames, outputfilenames, and programName.
frequency property has two possible values:
SimulationConfiguration::FrequencyType::CLK_4MHZ,
SimulationConfiguration::FrequencyType::CLK_20MHZ;

throttle property has a few possible values:
SimulationConfiguration::ThrottleType::REAL_TIME,
SimulationConfiguration::ThrottleType::FULL_THROTTLE,
SimulationConfiguration::ThrottleType::SINGLE_STEP;

vector<string> inputFileNames;
vector<OutputFile> outputFileNames;
where OutputFile is defined as

```
struct
{
        string filename;
        vector <string> outputnames;
        int period;
}
SimulationConfiguration::OutputFile;
```

The interface of this class consists of accessor methods for the properties:
void setFrequency(SimulationConfiguration::FrequencyType val)
SimulationConfiguration::FrequencyType getFrequency()

void setThrottle(SimulationConfiguration::ThrottleType val)
SimulationConfiguration::ThrottleType getThrottle()

addInputStimulusFile(string& filename): This file name later used when a new simulation session starts to create a stimuli generator.

addOutputStimulusFile(string& filename, vector <string>& outputnames, int period): These values later used when a new simulation session starts to create a stimuli recorder. such that the data is recorded into a file created with the given name. outputnames are the names or pins or ports.

3.12. Controller Class

This class adds a level of abstraction between initialization logic of user interface and manager classes. It allows us to separate input related logic from other components of our system. It implements the callback functions for the user interface. These callback functions

delegate the requests of user to manager classes after gathering the necessary input from the user. For example by displaying input dialogs.

startSimulator(): If the current project is already built, this function calls SimulatorManager's start function if not it warns the user that the current project is not compiled. When the simulation starts halt and pause simulation option is enabled.

haltSimulator(): This function stops the simulation currently running by SimulatorManager's halt function and reports failure or success message to the screen accordingly. When the simulation halts all irrelevant options in the menu bar are disabled.

pauseSimulator(): This function pauses the simulation that is running by SimulatorManager's pause function. It prints the paused line number on the screen. When the simulation pauses resume simulation option is enabled.

resumeSimulator(): This function resumes the paused simulation by calling SimulatorManager's resume function. When the simulation resumes halt and pause simulation option is enabled.

stepSimulator(): This function causes one line of source code is to be run by calling the SimulatorManager's step function.

addBreakPoint(): This function calls the addBreakPoint function of SimulatorManager with the source file name and the line number which the cursor is located.

removeBreakPoint(): This function calls the removeBreakPoint function of SimulatorManager with the source file name and the line number which the cursor is located.

addWatch(): This function displays an input dialog that prompts the user for memory location and calls the addLocationWatch function of the watchWindow class.

addInputFile(): This function displays an input dialog that prompts the user for a stimulus file name and calls the addInputFile function of the SimulatorManager class with the file name as a parameter.

removeInputFile(): This function displays an input dialog that contains a list of the current input files enabled for the current project and prompts the user to select a stimulus file name from the list and calls the removeInputFile function of the SimulatorManager class with the file name as a parameter.

adjustClockFrequency(): This function displays an input dialog that contains a drop down list that contains the possible clock frequency values and prompts the user to select one. Then it calls the adjustClockFrequency function of the SimulatorManager class.

addOutputFile(): This function displays an input dialog that prompts the user for a stimulus file name, sampling period in clock cycles, and the names of pins, special registers or memory locations, and calls the addOutputFile function of the SimulatorManager class with the file name and other inputs as parameters.

removeOutputFile(): This function displays an input dialog that contains a list of the current output files enabled for the current project and prompts the user to select a stimulus file name from the list and calls the removeOutputFile function of the SimulatorManager class with the file name as a parameter.

newProject(): This functions prompts the user for a file name and calls the createNewProject of projectManager class with the file name as a parameter.

openProject(): This function displays an input dialog that prompts the user for a project file name and calls the openProject function of the projectManager class with the file name as a parameter.

saveProject(): This function calls the save function of the projectManager class.

CloseProject(): This function calls the closeFile function of the projectManager.

addFileToProject(): This function calls the addFileToProject function of the projectManager.

removeFileFromProject(): This function calls the removeFileFromProject function of the projectManager.

buildProject(): This function calls the buildProject function of the compilerManager class.

CompileSingleFile(): This function calls the compile function of the compilerManager class.

disassembleFile(): This function displays an input dialog that prompts the user for a file name and calls the disassemble function of the compilerManager class.

uploadFile(): This functions prompts the user for a file name and calls the upload function of the Loader class with the file name as a parameter. This function shows a dialog to the user which states success or failure.

downloadFile(): This functions prompts the user for a file name and calls the download function of the Loader class with the file name as a parameter.

clear(): This function calls the clear function of the Loader class. This function shows a dialog to the user which states success or failure.

3.13. SimulatorManager Class

This class manages the simulator and its functions are triggered by the controller class. It has a subclass named simulationConfiguration.

start(): This function starts the simulation of a project which is opened in the GUI. It gets hex file name from simulationConfiguration class, and calls loadHexFile function of the simulator class with the file name obtained. Then calls getBreakPoints function of the simulationConfiguration class which returns us pairs of source file name and line numbers. It then calls the get corresponding address function if the source file and the line numbers obtained. Then it sets the breakpoints in the simulator by calling setBreakPoint function with the instruction addresses obtained. Then it calls getInputStimulusFileName of the SimulationConfiguration class with the stimulus file name as a string parameter. Then it calls setFileName of the StimuliRecorder class with the name of the input stimulus name as a string parameter. Then it calls getOutputStimulusName function of the simulationConfiguration class. Then it calls setFileName function of the StimuliRecorder class. The calls run function of the stimuliGenerator class. Then calls run function of simulator class and stimuliRecorder class.

halt(): This function halts the currently running simulation. It calls the simulationHalt function of the Simulator class. When the program is halted by simulator class it returns to simulationManager class and then this class prints the message to the screen

pause(): This function pauses the simulation that is running. It calls the pause function of the Simulator class. It supplies the paused line number for the simulationPause function of the Controller class.

resume(): This function resumes the paused simulation. It calls resume function of the simulator class.

step(): This function is triggered by Controller class and enables the user to run the program step by step.

addBreakPoint(): This function calls addBreakPoint of simulator class. It adds additional break points when the simulation paused.

removeBreakPoint(): This function calls the removeBreakPoint function of simulator class. It removes the previously located break point. The break point that is going to be cleaned is pointed by the cursor.

addInputFile(): This function calls the addInputStimulusFile function of the simulator class. It adds new input stimuli file for the project.

removeInputFile(): This function calls the removeInputStimulusFile function of the simulator class. It removes the selected input stimulus file from the input files of that project. It takes input file name as a parameter.

getNextBreakpoint( ): This function will be used for managing and finding the place of next breakpoint. While doing this, function will look up for the next breakpoint line by line. It takes the original debugger arguments which are the line addresses.

getLineNo(Breakpoint): This function gets the line number of the breakpoint that is set before simulation. This will help the application to use the breakpoint applications like stepping and pausing.

3.14. FileManager Class

This class manages the file operations. It has methods that handles file operations such as: Opening a file, closing a file, adding a file to a project, creating a file in a project, saving a file.

closeFile(): Every file will have a unique ID of integer type and given this information closeFile will close the source file which was opened by the editor. This function returns nothing. It prints failure message in case of an error. It is triggered by the GUI class.

---

addFile(): This method takes the file name as a string parameter. Its function is adding a source file to an opened project. After the operation finishes successfully it returns or prints nothing but on the contrary if it fails error message is printed.

close(): This method closes the currently open file on the editor and file ID is given as an integer parameter to it. Incase of an error it prints failure message, otherwise it neither returns nor prints.

create(): This method creates a new file in the project. It assings a unique ID to the created file and returns nothing. Prints error message in case of a failure. It is triggered by the GUI class.

open(): Given a file name as a string this method specifies the file ID and opens the file. Incase of an error it prints failure message, otherwise it neither returns nor prints. It is triggered by the GUI class.

Save(): Given the file ID this method saves the specified file and returns void or prints error message in case of a failure. It is triggered by the GUI class.

3.15. EditorComponent Class

This class represents the text editor of the software. It has methods to handle text editor operations. It checks whether the contents of a code segment in the editor changed. Also, it saves the contents of the editor. It first checks whether a file is saved or not before closing the file. When tis function finishes its instances must be destroyed.

checkChanged(): It checks whether there is any change or not in the file whose ID is given as an integer parameter and it returns Boolean. If the file opened in the editor part has changed it returns true otherwise false. It is trigered by the FileManager class.

dispose(): It takes no argument and returns void. It checks whether the process has finished and if so it closes the currently open file. It is trigered by the FileManager class.

createBuffer(): It creates a buffer to write new text. It stores the characters and these will be recorded by other functions. It returns void. It is trigered by the FileManager class.

setContents(): It gets the saved files contents whose id is given as integer parameter and sets this contents to the buffer. It takes char[ ] to store buffer and also takes contents of the file which is of string type. It returns void. It is trigered by the FileManager class.

getContents(): It gets the contents of the editor, whose id is an integer type, to save. It returns void. It is trigered by the FileManager class.

3.16. Project Class

This class handles the operations of project that is developed. It opens a project and handles the operations of the project files. When a project is closed it checks whether the files in the project is saved or not. It removes files from the project or adds files to the project.

checkChanged() : It checks whether there is any change or not in the project that is currently open and it returns Boolean. If the project opened in the editor part has changed it returns true otherwise false. It is triggered by ProjectManager class.

write(): It writes to the currently opened file of the project. This method takes the file ID as an integer parameter. It returns void. It is triggered by ProjectManager class.

removeFile(): It deletes the file ,whose name is given, from the project. It takes the file name as a string parameter and returns void. It is triggered by ProjectManager class.

addFile( ): It adds the file whose name is given from the project. It takes the file name as a string parameter and returns void. In case of a failure it prints error message. It is triggered by ProjectManager class.

load( ): It loads the project file who is given as a string parameter to the project. It returns void. In case of a failure it prints error message. It is triggered by ProjectManager class.

3.17. StimuliRecorder Class

This class is for output operations. It records the results of a run and specifies the output file name.

recordOutputToFile( ): This function records the generated output of the requested stimulus to the file. The recorded output will be also kept in this class and will be executed during the simulation cycle1.

setFileName(fileName): Like in StimuliGenerator Class This function sets the FileName as the file name of the output stimulus file.It takes a String as an argument.The file name that

comes from SimulatorManager Class will be set to this class as an argument as a request during setup operations.

run(): This function runs the application. The request comes from the SimulatorManager Class.

3.18. SimulatorGUI Class

This class is for redrawing the interface for simulation. When it is executed it refreshed the simulation. The newly predicted simulation comes as an interface to the user.

redraw(): This function draws the simulation on the screen. The drawing application is simple. This function will work with the pipes.StimuliPipe2 provides the expected interface configurations and this will be written to StimuliPipe1 by the help of the write() function during he simulation cycle 1.

3.19. MemoryContentDisplay Class

This is for displaying the contents of the memory locations. It has a method for refreshing the situation of the memory locations.This class will take part during the simulation cycle 2.

read(): This function shows the contents of the memory locations. It works under the guidance of PicProgramMemory by reading the contents of memory locations and taking them back to MemoryContentDispla y Class. It takes integer offset as argument. This function takes part during the simulation cycle 2.

3.20. StimuliFile Class

This class is for storing the stimuli file and its configurations. After loading hex file and setting all the breakpoints so far, loading the stimuli file will be needed for application .The threads has to be run independently from the simulation cycles .For this purpose an outer execution takes place to construct a healthy stimuli file.

3.21. StimuliGenerator Class

This class is for input operations. It reads input from the input stimulus and sets the input file name and starts giving input during the run of the program.

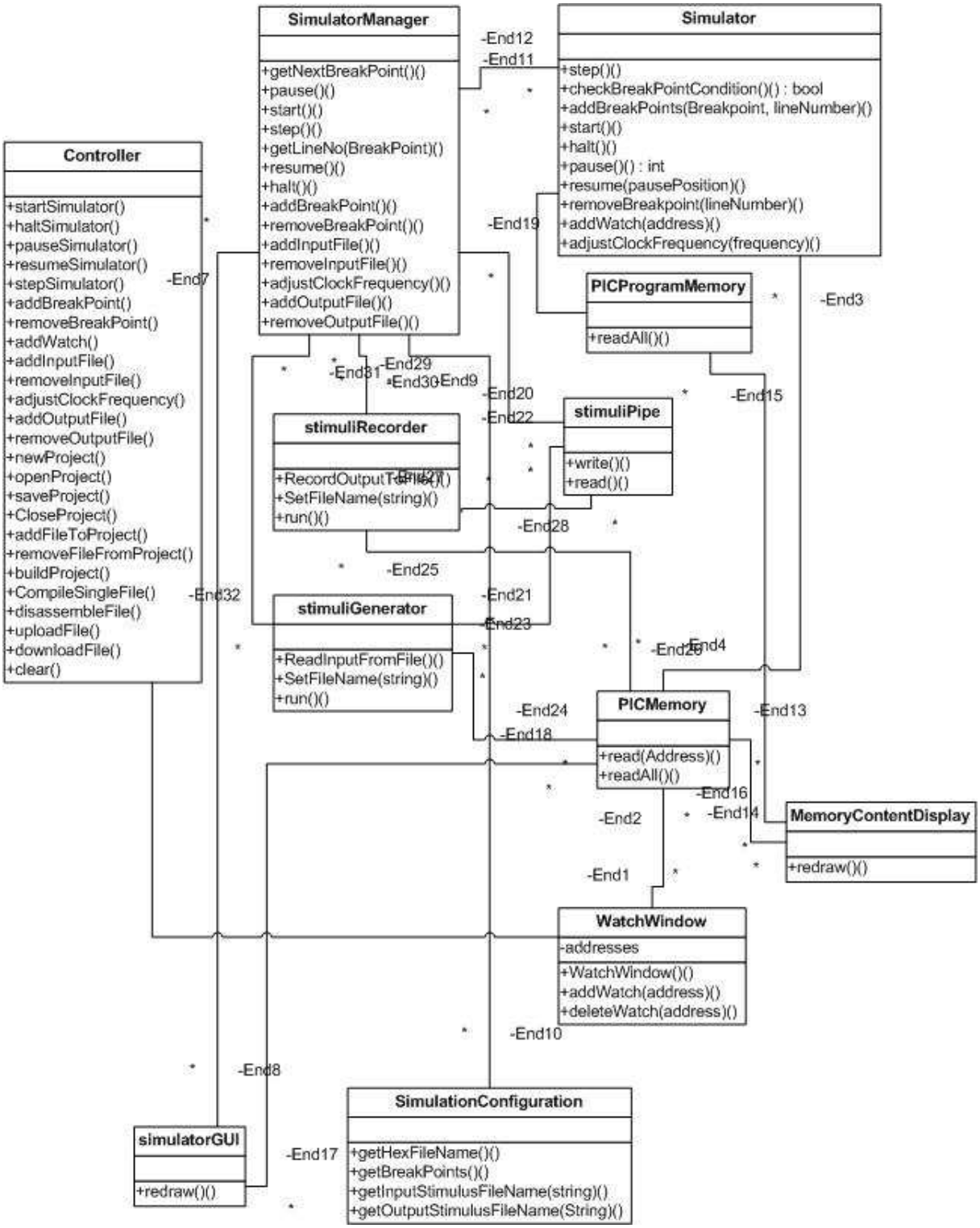
readInputFromFile(): It reads the stimulus from the file.

setFileName(): It sets the fileName as the file name of the currently generated stimulus file.

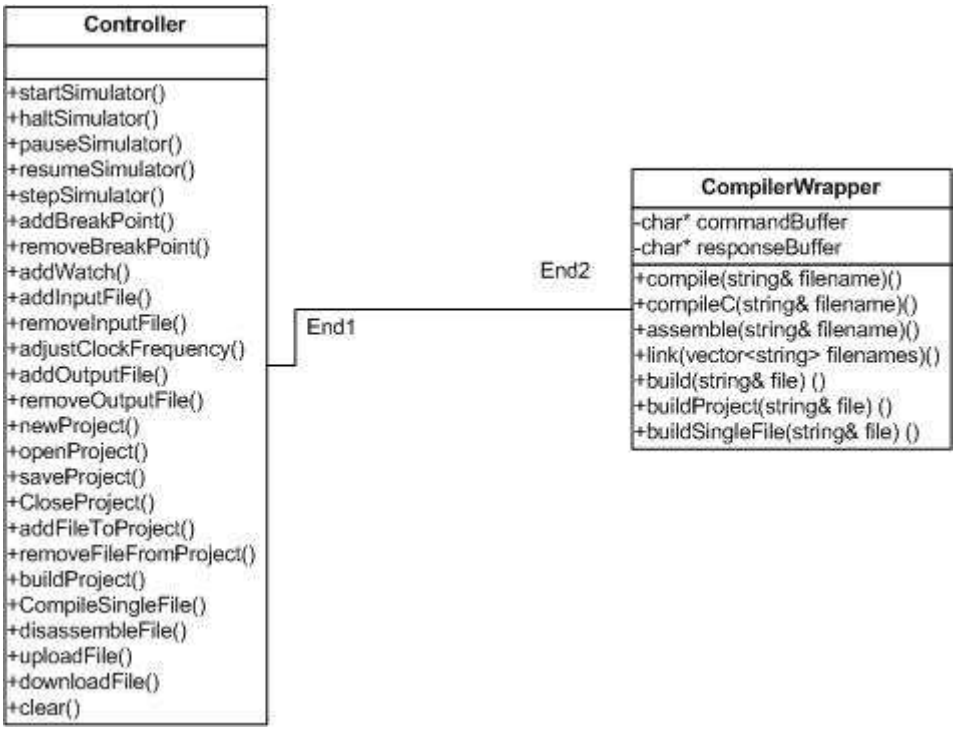
run(): It runs the simulator according to specified configurations by the user.

4. MODULES

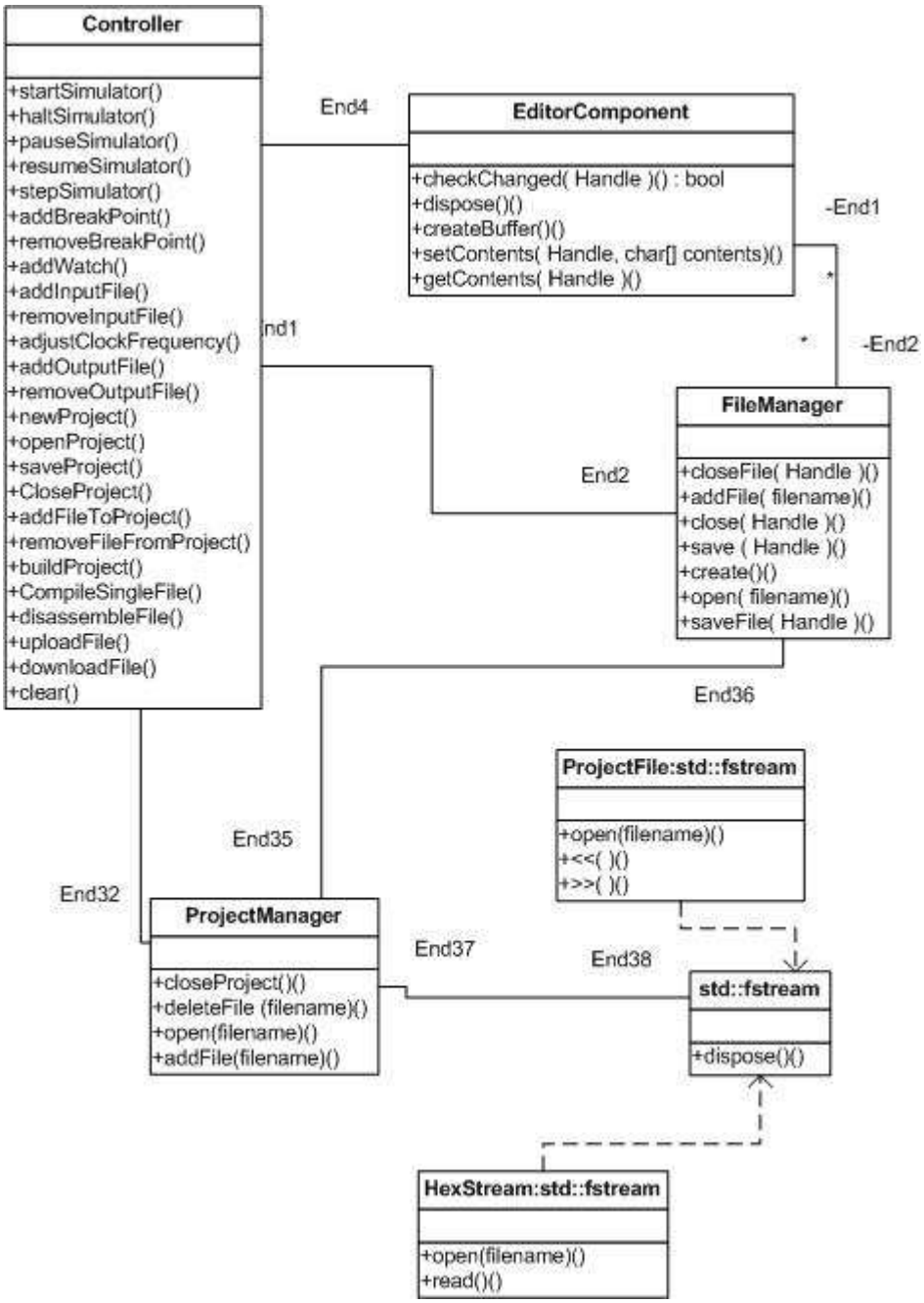
4.1. Simulator



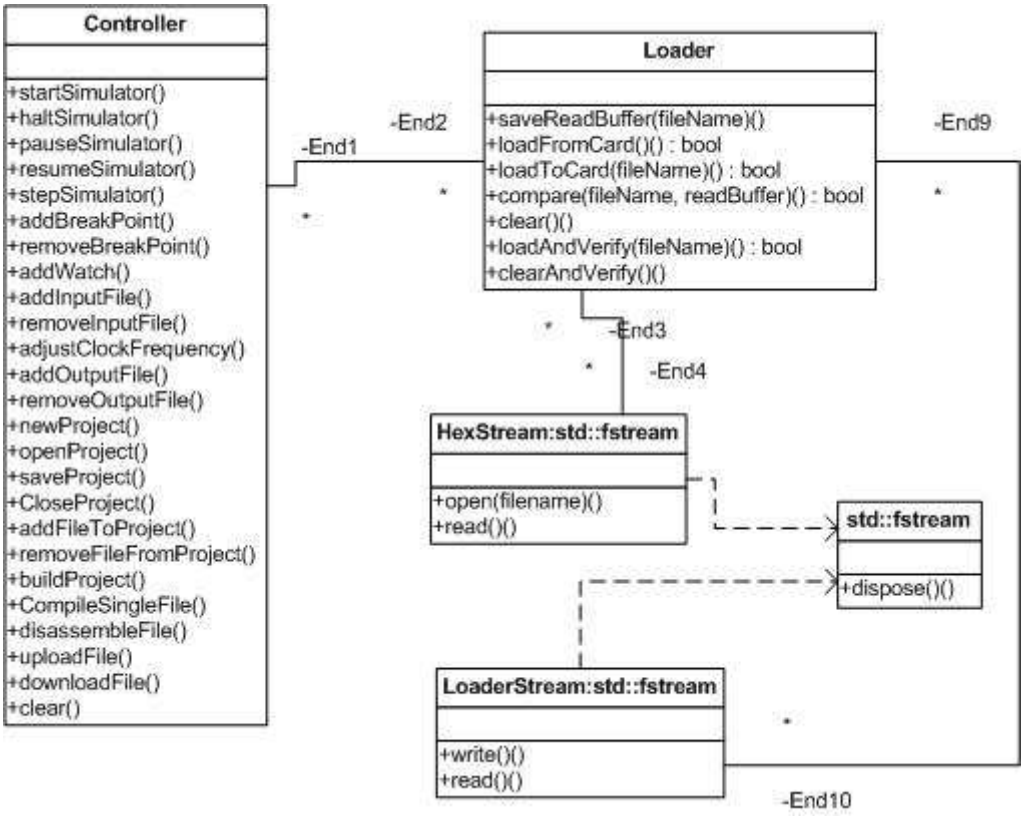
4.2. Compiler


Controller
+startSimulator()
+haltSimulator()
+pauseSimulator()
+resumeSimulator()
+stepSimulator()
+addBreakPoint()
+removeBreakPoint()
+addWatch()
+addInputFile()
+removeInputFile()
+adjustClockFrequency()
+addOutputFile()
+removeOutputFile()
+newProject()
+openProject()
+saveProject()
+CloseProject()
+addFileToProject()
+removeFileFromProject()
+buildProject()
+CompileSingleFile()
+disassembleFile()
+uploadFile()
+downloadFile()
+clear()
End1
End2
CompilerWrapper
-char* commandBuffer
-char* responseBuffer
+compile(string& filename)()
+compileC(string& filename)()
+assemble(string& filename)()
+link(vector<string> filenames)()
+build(string& file) ()
+buildProject(string& file) ()
+buildSingleFile(string& file) ()

4.3. Project/File



4.4. Loader



5. FILE FORMATS

5.1. Project File

project file contains the names of the files included in the project, as well as input file names and output file specifications.

specification

default file extension for prject files is ".prj"

each file begins with a number on a line by its own that indicates the number of source files included in the project. which is followed by that many lines, with each source file name on a separated line. which is then followed by a number on a separate line which indicates the number of input files, followed by that many lines each containing the name of an input file

name. input file names are followed by a number that indicates the number of output files to be created. each output file specification begins with name of the file to be created followed by the period to be used in 'simulator trace' in number of instructions cycles, and then by the number of output signals(pins or ports). this is followed by that many ouput signal names (each of which may be a pin name a special function name or a hexadecimal address of a memory location) seperated from the previous ones by white space and ending in a new line.

example project file contents:
myProject.prj

```
----------------------------------
3
main.c
myHeader.h
myFile.c
4
ramp.stm
sawtooth.stm
toggle.stm
test.stm
2
out1.stm 1 100
b0
out2.stm 3 250
b1 portc 0x01EA

----------------------------------
```

5.2. HEX file

hex file is used to store binary information in a form that is conducive to input from or output for human users, such as a console or a text editor. the standard is developed by Intel. There are many hex file formats actually. We are using the most basic one; the 8-bit format in our project, which is generally used by other PIC development systems as well.

specification

default file extenstion for a HEX file is ".hex". A hex file is a text file that consists of lines of text. Each line is called a record. each record starts with a colon(':') and and in a new line character. Between the two there are a number of pairs of eponymous hexadecimal digits. first of those pairs is the number of data bytes in the record. The following four digits is the 16-bit offset value in the address space. Then followed by two hexadecimal digits that determines the record type. We will only use two types of records. A data record (signified by 0) and end of file record (signfied by a 1). This is followed by data bytes of the given number. The last of bytes is a check sum.it is the two's complement of the sum of all the previous bytes.

5.3. Code File

A Code File is a special kind of object file, that is it holds machine instructions and symbol table information. The distinguishing feature of Code File format is that it also provides a way to indicate from whence a particular instruction comes. Namely, given a specific machine instruction in a program we may find the source file and line number it corresponds to, and vice versa. This information is particularly needed for debugging purposes. Breakpoint operations of source level debugging needs the information mentioned above.

specification

default file extension for stimulus files are ".cod". Code File specification is developed and maintained by ByteCraft Ltd. A detailed specification may be obtanied by contacting the firm through the URL http://www.bytecraft.com/codfile.html

The full description of the format is beyond the scope, but basically a COD file consists of 512 byte records. Some of these records are directory records while others are data records. By parsing this file we may obtain the content of the program memory, list of the source files used and the global and local symbols, etc. the file also contains debugger directives that are embedded in the source code by means of #pragma constructs within the source code. However our system will not support this feature.

5.4. Stimulus File

In our project, stimulus files are used for input to and output from our simulator. Stimulus files may be used for testing and program verification purposes. Our stimulus file format is developed to be human readable, as well as easily parsable by scripting languages such as perl. Thus instructors may provide students with sample I/O files to ensure that students have learned a PIC subject well enough. Or an TA might write a script to evaluate assignments.

specification

default file extention for stimulus files are ".stm".

We planned our stimulus files to be compatible with those of mplab. However we could not find a full official specification. So, we have devised our own specification, based on "MPLAB IDE Simulator, Editor User's Guide" reachable at (http://ww1.microchip.com/downloads/en/devicedoc/51025e.pdf) and by examining stimulus file examples we had found, used by various simulators. Stimulus files support line comments and semicolon(';') is the comment character. thus all text, starting with a semicolon upto an including the next newline character is ignored. First input line consists of the word "cycle" or "step" followed by a list of pin names, special function register names, and register addresses. register addresses are in hexadecimal format. Each following line consists of a time amount followed by a list of values, each separated rom the previous by whitespace.time amounts are in decimal format. If the file starts with the keyword "cycle" then the time values represent the number of cycles passed from the time the stimulus generator was reset. Similarly, if the file starts with the keyword "step" then the time value in the first line represents the time passed since the stimulus generator was reset. the time values represent the number of cycles passed since the time the previous line of stimulus was actuated. Pin values are in binary format (that is '1' or '0') register values are in hex format.

Finally, a stimulus file may end in the keyword "reset". which means the counter of the stimulus generator is reset for that file and the file is actuated over and over until the stimulus generator is stopped.

6. Development Environment

Our project will be implemented using Dev-C++, which is an IDE built on MinGW compiler. All group members are familiar with it. Dev-C++ also has support for CVS. A group member will be responsible for the consistency of interface of the project. Source file names and locations will be frozen early in the implementation. Also the as much of the interface declarations of program modules as possible will be placed in header files which will be kept by the mentioned group member which will keep all group members infromed about the changes of the interface of the modules. no great overhauls of the organizations of modules will be allowed.

As our preliminary tests showed that windows port of the POSIX threads library is not fast or reliable as Win32 threads, Win32 API will be relied on to provide multithreading. However system dependent code will be seperated from the rest of the program by a layer of abstraction as to make the program more robust in the face of any API deprecation and allow easier porting to other systems.

FLTK is a lightweight and fast widget toolkit. It can be statically linked with programs and results in relatively small size executables. Any GUI component not coded in Win32 API will be implemented with this toolkit. It is system independent, and has a clean -easy to use- interface.

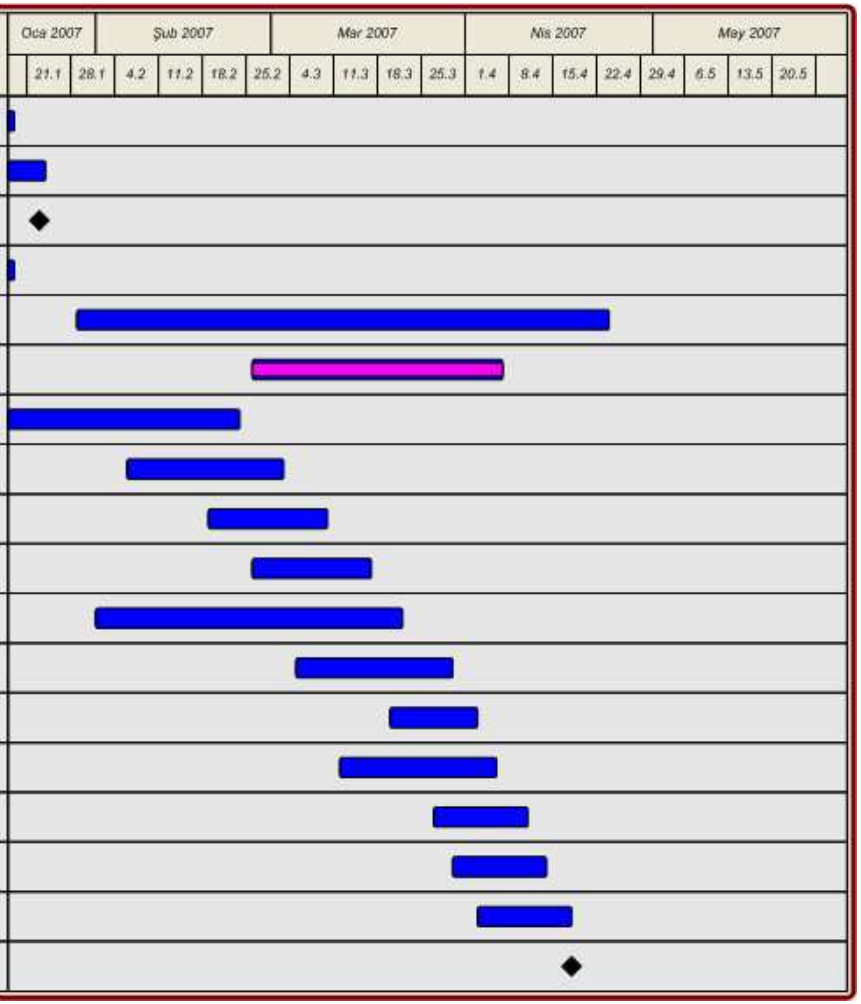
Deployment

A makefile will be maintained as to allow testing by developers to ensure that the program compiles at each step of the development. Makefile will be used to build development and release versions for the program. The Makefile will locate the executable in a specified location, and all other files that will be required during excution will also be located to a fixed location with respect to the executable. All code except for the third party software and possible future pluggable extensions (none official yet) will be linked statically. All

dynamically linked code will be located in a fixed position relative to the program executable and will be explicitly loaded from that location, so as to prevent any version conflicts of the dynamically linked libraries.

7. APPENDIX

ID	Task Name	Start	Finish	Duration
1	**Prototype**	18.01.2007	18.01.2007	1d
2	Prototype Implementation	18.01.2007	23.01.2007	4d
3	Prototype Demo	23.01.2007	23.01.2007	0d
4	**Implementation**	18.01.2007	18.01.2007	1d
5	Implementation of GUI	29.01.2007	23.04.2007	61d
6	Implementation of Basic Components	26.02.2007	06.04.2007	30d
7	Implementation of Disassembler	18.01.2007	23.02.2007	27d
8	Implementation of Simulator	06.02.2007	02.03.2007	19d
9	Implementation of Assembler	19.02.2007	09.03.2007	15d
10	Integration of Modules Phase-I	26.02.2007	16.03.2007	15d
11	Implementation of Compiler	01.02.2007	21.03.2007	35d
12	Implementation of Loader	05.03.2007	29.03.2007	19d
13	Implementation of Project Manager	20.03.2007	02.04.2007	10d
14	Integration of Modules Phase-II	12.03.2007	05.04.2007	19d
15	Implementation of Simulator Stimuli Generator and Recorder	27.03.2007	10.04.2007	11d
16	Implementation of Simulator Extensions: A/D Converter	30.03.2007	13.04.2007	11d
17	Integration of Modules Phase-III	03.04.2007	17.04.2007	11d
18	**First Release**	18.04.2007	18.04.2007	0d



19	**Testing**				
20	Unit Tests				
21	Tests				
22	**Improvements**				
23	Implementation of Simulator Extensions: Sound				
24	Implementation of Simulator Extensions: USART I/O				
25	Implementation of Simulator Extensions: SMART CARD Reader				
26	Implementation of Simulator Extensions: USB				
27	**Documentation**				
28	Help Files				
29	Sample Demos				
30	Internet Documentation				
31	**Final Release**