Test Specification Report

DEVEMB

by

ResolveSOFT

Fatih Mehmet DOĞU

Hayri Çağlayan ERDENER

Adem HALİMOĞLU

Ulaş TUTAK

May 7, 2007

TABLE OF CONTENTS

1 INTRODUCTION

The purpose of this Test Specification Report is to specify the scope, strategies and plan of testing that is going to be performed in the remaining time in the project life-time of our project DEVEMB. In general, during the development process of our project, functionalities are tested by their developer and by the other group members in a usual way of developers. However, we need a standardized testing process anymore since our project is getting larger in the course of time. As a result of this, in this document our aim is to check and provide the consistency of our project's modules and the communication of our project modules when they are integrated to realize our main goal which is providing an error-free and well built project to the users.

1.1 Goals and Objectives

After generating the source code, software has to be tested to reveal and correct bugs as many as possible before delivering it. Our goal is to design a number of test cases that have high possibility of finding bugs.

While we are testing our project, our main goals are as follows:

- Finding errors and correcting them in as possible short time as,
- Guaranteeing that our product satisfies the user requirements,
- Guaranteeing that our product provides all of the functionalities that are committed previously by us,
- Guaranteeing that our product has high performance,
- Demonstrating that our product contains minimum number of bugs.

1.2 Scope of the Document

This document is prepared for specifying the official description of the testing phase of DEVEMB. Testing plan is designed to check the requirements, correctness and performance issues. Our testing process consists of integration testing, unit testing, validation testing and higher order tests.

We will probably spend the biggest part of our testing time on unit testing because we have many classes and we will test each class one by one although each function or class is tested by its developer(s) during the implementation of it whether it works correctly or not.

The most important part of our test phase is integration testing and most of the time will probably be spent for this testing because there are many modules and we use some programs that we did not implement such as gpasm, sdcc in our software. The communication and harmony between modules have to be tested to prevent unpredictable consequences.

For higher order testing, we decided to make Performance Testing, Stress Testing, Alpha and Beta Testing. The detailed information about these testing types and how we will perform them is explained below in the related part.

Validation testing for our project is going to be including requirements validation and design validation subparts that are explained in a detailed way in the following parts of this document.

2 TESTING STRATEGY AND PROCEDURES

2.1 Unit Testing

2.1.1 SimulatorGUI Class:

This class is responsible for drawing the PDB on the screen, showing the changes on it sent by Simulator Class and sending some impulses by its switches. It is important for this class is providing high refreshing rate, showing the effects coming from Simulator Class and transmitting the user requests correctly and quickly.

LEDs and Seven-Segment-Displays: For its functionality tests, our SimulatorGUI Class will be tested whether it displays the data coming from the Simulator Class on the LED, seven-segment-displays or not. For this purpose we assign several different binary numbers to PortA, PortD and PortE, and observe whether the SimulatorGUI displays what it must display.

Switches: Switches will also be tested by using different cases. For example we again assign a number of different binary numbers to PortB, TrisB and OPTION_REG registers and control the values that are exists in RB0, RB1, RB4, RB5 before and after pressing and

releasing the switch. Because for example, the existing value in RB0 before pressing SW0 is assigned to the RB4, after pressing SW1 RB0's value is assigned to RB5, after pressing SW4 RB1's value is assigned to RB4 and finally after pressing SW5 RB1's value is assigned to RB5. All of these cases will be tested by using different binary number that satisfies the situations stated above.

Refresh Rate: Refresh rate will also be tested but for this test we have got any formal parameter, we only test it by just looking at SimulatorGUI. If it is satisfactory, test will be successful.

Lcd: While testing LCD, we assign a number of different binary numbers to PortD and PortE. Then, we control the LCD whether it displays what it has to display. After that, we save some data on the LCD's memory. Then, we control the LCD again, if it displays the correct things, our test will be successful.

2.1.2 Project Class:

Project Class is responsible for operations that may be come from user such as adding and removing file to the currently opened project, loading previously written project and writing the currently active project to a file. Moreover, it is also responsible for checking the changes on the project.

Adding File: To be able to test this functionality of Project Class, we will try to add file names by using addFile(const char* fileName) method of this class and by using any number of different file names. Then we will control the content of fileName vector, in which file names are saved, whether the file names that we give as a parameter to the method addFile(const char* fileName) exist or not. If that file names exist in the fileName vector, the test will be successful. Moreover, we will try to add an existing file name to our vector, and then we will control the fileName vector whether that file name exists once or twice. If it exists once, the test will be successful.

Removing File: For testing this behavior of our class, we will try to remove file names that previously added to the fileName vector by using removeFile(const char* fileName) method of this class. Then we will control the content of fileName vector whether the file names that we give as a parameter to the method removeFile(const char* fileName) exist or not. If that file names do not exist in the fileName vector, the test will be successful. Moreover, we will

try to remove a nonexistent file name from our vector. If it turns a warning, the test will be successful.

Loading Project: In order to test this loading process, we will try to load a project that is previously written by using load(const char* fileName) method and any file name that exists. After that we will control the contents of the fileName vector, if the file names are loaded correctly, test will be successful. Besides, we will try to load a nonexistent file and control the result, if our program returns a warning, test will be successful.

Writing Project: For this functionality of Project Class, we will add a number of file names to currently opened project. Then some of the file names will be removed. After that we will try to write the project by using write(const char* fileName) method and any file name. Then it will be checked whether it is able to write the project correctly or not. If it writes the project accurately, our test will be successful.

Changes Checking: For the purpose of testing this, we will check the initial value (it should be False) of the boolean variable changed. Then, we will control this variable after adding a new file name to the project. If its value is True, test will be successful. This procedure will be applied for removing and loading functionalities. If after all these operation the value of variable changed is True, our test will be successful.

2.1.3 ProjectWindow Class:

ProjectWindow Class' task is to display the project properties and its components such as its file names. This class displays the data come from the Project Class on the screen.

While we are testing Project Class, we will test the ProjectWindow Class at the same time since these two classes are strictly related with each other, one shows the data on the screen comes from the other. Therefore, after testing each functionality of Project Class, we will control the ProjectWindow whether the changes made on the Project Class seen or not. For all Project Class tests such as Adding File, Removing File, Loading Project, Writing File, if changes can be seen on the screen, tests for ProjectWindow will also be successful.

2.1.4 WatchWindow Class:

This class of our project has responsibility about displaying the contents of the specified memory addresses. This class is important for testing and tracing the program.

Adding Watch: In order to test this function, we will try to add addresses by using addWatch(int address) method of this class and by using any number of different addresses. Then we will control the content of addresses vector, in which addresses are saved, whether the addresses that we give as a parameter to the method addWatch(int address) exist or not. If that addresses exist in the addresses vector, the test will be successful. Moreover, we will try to add an existing address to our vector, and then we will control the vector addresses whether that address exists once or twice. If it exists once, the test will be successful. Besides, we will randomly generate 8-bit binary numbers then write this numbers to the debugging window. After that, we assign this number to a register and we refresh the window. Then, we show the value on our trace window. If the values on the debugging window and trace window are the same, our test will be successful.

Deleting Watch: While testing this behavior of WatchWindow class, we will try to delete addresses that previously added to the addresses vector by using deleteWatch(int address) method of this class. Then we will control the contents of the addresses vector whether the addresses that we give as a parameter to the method deleteWatch(int address) exist or not. If that addresses do not exist in the addresses vector, the test will be successful. Additionally, we will try to delete a nonexistent address from our vector. If it turns a warning, the test will be successful.

2.1.5 FileManager Class:

FileManager Class deals with the file operations such as creating, opening, closing and saving file, adding file to a project, etc.

Opening File: Testing of this functionality will be performed from the GUI again. If specified file whose name selected from the open dialog box will be opened successfully, our test will be successful. This process repeated a number of times, if all them successful, the test will be successful.

Closing File: Before testing of this, a few files will be opened. After that, some of them will be tried to close by using the editor's close facility. If close operation for all these files are successful, our test will be successful, too.

Creating File: For testing this functionality, File->New button pressed. Then if a new blank file is open on the editor, test will be successful.

Saving File: As, in Closing File, before starting to the testing, we will open a file. After that, we will make some changes on that. Finally, we will press the File->Save button from the editor. Then we will control whether the file is saved correctly or not after by comparing currently opened file on our editor and saved file which is open by another editor. If they are the same, our test will be successful.

2.1.6 Simulator Class:

In each of the following test cases when program state is said to be recorded it is meant that memory contents, program counter, and stack contents are written in single column hexadecimal format in a text file.

Initial Set-up Procedure

- 5 programs that does not halt, that is that has a main loop that is reasonably long, selected from the learning material for pic assembly programming, such as recitations, homeworks or the like.
- Programs are started. If any program fails to run or behaves abnormally in an unforeseen manner the event is logged. And the program changed with another one and this procedure is repeated until 5 reasonably working programs are obtained.
- Programs are run for 10^i (i = 1,.., 6) steps with no input and program state is recoded in a file. Similar is done with gpsim, which supports cycle breakpoints, and results compared.
- Program is run for 5 seconds and program state, as well as number of cycles program has run, is recorded.

Pause-Resume Functionality

For each of the 5 test programs:

- The test program is run in a separate thread as in the last case for the number of steps recorded in the second part.
- During execution, every 10 ms, the program is paused and resumed.
- After the completion of the predetermined number of steps the program state is recorded.
- Any anomalies encountered, as well as discrepancies as discovered by text file comparison programs in addition to human inspection, are noted.

Halt and Reset Functionality

For each of the 5 test programs:

For each i (i = 1, ..., 6), as in the first part.

- The test program is run in a separate thread as in the last case, for 10^i cycles.
- Then the thread will be suspended, and program state will be recorded. Then halt() procedure will be run and program state will be recorded. Then reset() procedure will be run
- For the current i; there will be three versions of program state: from first test case, before halt and after halt. Comparison as done in the previous case will be repeated between the program states of first test case and before halt; and between the program states before halt and after halt.

2.1.7 HexFileCodec Class:

Encode Functionality

- A hex file will be imported to MPLAB and program memory contents will be exported as a text file.
- A driver class will be written that will read in the values from the text file and encode it another hex file.
- This second file will be imported to MPLAB and exported as a text file as before.

- The two text files exported from MPLAB will be compared.

Decode Functionality

- A hex file will be imported to MPLAB and program memory contents will be exported as a text file.
- A driver class will be written that will read a hex file and decode in a buffer and will read in the values from the text file and then compare the two values read from buffer and read from file and report discrepancies.

2.1.8 PicMemory Class:

Special register functionality

Driver class will be written that will test indirect access, memory bank switch, and computed jump related functionality of the memory component.

Memory Bank Functionality

Basic procedure:

- Generate a random 7-bit address, called *addr* hereafter, in the general purpose registers'
- Address range.(0x20, ..., 0x7f) generate 4 random 8-bit values, called v_0, v_1, v_2, v_3 respectively hereafter. Generate a random orderings (shuffle) of the numbers 0,1,2,3. Write the generated random numbers to location *addr* in the banks in the randomly generated order, so that value v_i goes to bank i. generate another random order. Read the values at location *addr*, in the generated locations, where value read from bank j is called rv_j.
- For k (k=0, ..., 3) compare v_k with rv_k.

Indirect Read and Write Functionality

Basic procedure

- Generate a suitable (general purpose) address randomly. Generate an 8-bit value

randomly. Attempt an indirect write to the location by using writing the lower order 8 bits

- Of the address generated to the FSR and writing the highest bit of the address generated
- To IRP(STATUS<7>); and writing the value to INDF (address 0x000). Now read the value directly by changing to the relevant bank by setting RP1 and RP0 (status<1:0>) to
- The higher two bits of the address generated; and reading the value at location denoted by the lower 7 bits of the address generated. Compare with the values generated.

Additionally:

i) Repeat the basic procedure for 1000 times

ii) Repeat the basic procedure for 1000 times; with reversing reading and writing operations to test indirect reading

iii) Repeat the basic procedure; using i (i=0x000, ..., 0x1ff) for address and j (j=0x00, ..., 0xff) instead of random numbers for value.

iv) Repeat the above variation with reversing reading and writing operations to test indirect reading.

Compare values in each case, paying special attention to boundary conditions where address=0x000 etc. Comment on the compliance with the standard behavior.

Program Counter Related

Computed jump

Generate random 14-bit addresses. Write the upper 5 bits to PCLATH register. Write the lower 8 bits to PCL register. Compare program counter with the address generated.

Generic Read Write Performance

Basic procedure:

- Write driver class that runs a thread which runs an infinite loop that makes consecutive reads from general purpose registers' range and increases a counter.
- In the main thread after 5 seconds check the counter.
- Compute average frequency and average time spent for a read.

i) repeat the basic procedure for writing instead of reading

ii) repeat the basic procedure displaying the computed frequency every 10 ms while another application running on the background.

Comment on the consistency of the frequency of memory read times.

iii) repeat the above variation for writing instead of reading

2.1.9 MemoryContentDisplay Class:

Write a driver class that will:

- Fill the pic memory with randomly generated 8-bit values export the memory contents to a text file in MCH single column format.
- Import the file to MPLAB and load starting from address 0x000.
- Display the contents of the memory using MemoryContentDisplay class compare the two displays visually (about 512 entries).

2.1.10 SimulatorManager Class:

Clock Frequency Functionality

Basic procedure:

For clock speeds 4 MHz and 20 MHz:

- Run for 10 seconds with programs obtained for Simulator class test
- Count actual cycles executed.

- Compare with theoretical values.

Full Throttle Execution Performance

Write driver class that runs a thread which runs an infinite loop that runs the simulator at full throttle and increases a counter each cycle.

In the main thread after 5 seconds check the counter. Compute average frequency and average time spent for a cycle.

2.1.11 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. In this manner our loader class has to work properly. To test this method, we have to create functions that loader includes. To do this we load our program and test if the program works properly, from simulation of the program. Seven segment displays and leds will assure us that the hex file loaded properly .We also try to exceed the paging problems because pic takes 8kb of data from file.

2.1.12 EditorComponent Class:

Editor component requires simple module testing. This component hast to hold the file that the user write delete or update. Before executing and loading the program into buffer, as usual the editor component has to save the new version. It has to be checked whether this version is appropriate or not.

2.1.13 CompilerWrapper Class:

This class provides a wrapper for shell commands that will be run in the course of compilation related actions. After compiling the program it has to be run on debug mode taking step by step action for all modules of implementation. 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.

The compiler module has specific types of function alignment. To test simple c compiler compiling the file in debug mode is appropriate action.

2.1.14 Breakpoint Class:

To test whether this class works or not looking the contents of Breakpoint object after creating it with Breakpoint(string source, int lineNo) is appropriate. This method stores the line and address of the breakpoint. It has to be run in debug mode step by step after setting the breakpoints. If the debugger stop in specified position and stores the address of that, breakpoints work properly.

2.1.15 StimuliPipe Class:

After writing the code, to see the results of the written program or to trace the program that we had written 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. 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. To test write

function of stimuliPipe Class we will execute simulator and check whether a stm file which has the given file name during execution and time and situation of leds. To test read function we will write a stm file and execute the simulator to see whether it is executing according to the given stimuli file commands or not.

2.1.16 StimuliGenerator Class:

StimuliGenerator produces some records to perform the simulation process. 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. It has functions:

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.

To test setFileName function we will execute the simulator and check whether filename and name of the currently generated stm file are same or not.

To test run function we trigger the programme from the GUI and check whether it reponds or not.

2.1.17 StimuliRecorder Class:

StimuliRecorder records the output of the simulation process. This class is for output operations. It records the results of a run and specifies the output file name. It has functions:

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.

To test recordOutputToFile we execute the simulator and give input file then check whether output is normal or not.

To test setFileName as in the StimuliGenerator Class we control whether the FileName is set as the filename of the output stimilus or not

To test run we check whether the class responds to the request that is coming from simulatorManager class.

2.1.18 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 have to be run independently from the simulation cycles. For this purpose an outer execution takes place to construct a healthy stimuli file. To test stimuliFile class we will control the change of GUI during execution and the result stored in the stm file and also we will check whether the breakpoints are signed in the text correctly or not by looking at the stm file.

2.1.19 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. Output names are the names or pins or ports.

To test setFrequency after execution of the simulator we check the value of the frequency that is recorded in the stm file and the value we had given as input.

To test setThrottle and getThrottle after execution of the simulator we check the effect of the turn to the output stm file and the actual angle that we turned the throttle.

To test addInputStimulusFile we control whether the related input stm files are in the project folder or not.

To test addOutputStimulusFile we control whether the related output stm files are in the project folder or not.

2.2 Integration Testing

Since, our project is composed of twenty-one classes, four modules and we integrate a few different programs that are not implemented by us, they have to interact, communicate well with each other so that the resultant product works properly. Sometimes the product does not work properly after integration phase although individually integrated classes, modules or integrated programs works correctly. In order to ensure that our product executes as it should be after integrating all modules, we determined a testing policy.

As the ResolveSoft group, we will use bottom-up integration test method. After implementing and doing unit tests classes are integrated to their modules and sometimes if need, we integrate some other programs. After these integrations the modules will be tested using bottom-up integration testing wholly whether the whole system works correctly or not. For the unit testing, we said that we would test each class individually. However, for integration test, while we are testing a class, we will also test all classes related with that class. Thus, if any class is changed, while we are testing it after changes, we will test all the related classes because a change made in any class may affect not only that class but also other related classes with it.

Sample Integration Test Scenario: This sample scenario is prepared for after integrating the Assembler Class, Compiler Class, EditorComponent Class, FileManager Class, Simulator Class and SimulatorGUI Class.

Open a new project by clicking File->Project->New

Add files to the project by clicking File->New

Make some changes on the files

Save the file by clicking File->Save or save the project by clicking File->Save All

Compile project by clicking Execute->Compile

Assemble the output file of the compilation by clicking Execute->Assemble

Load the output file of the assembler to the simulator by clicking Simulator->Load

Watch the resultant output of the project from the SimulatorGUI.

2.3 Validation Testing:

The purpose of the validation tests are to ensure that the implementation of the 'ResolveSOFT DEVEMB software', also called 'the program' in this section, follows the requirement and design specifications previously stated in "ResolveSOFT Requirement Analysis Report", also called 'analysis report', and "ResolveSOFT Final Design Report" documents, respectively. To this end, we will conduct two groups of tests that reflect this situation; namely 'requirement validation tests' and 'design validation tests'.

2.3.1 Requirements Validation:

2.3.1.1 Interface Requirements Validation

Because testing graphical user interfaces exhaustively is very labor intensive, and because we have neither the manpower nor the time to indulge in extensive interface testing, we limited greatly the scope of interface testing with respect to that of an industrial grade project, assessing that same constraints also apply to other DEVEMB project groups.

In order to verify that the program complies with the interface requirements tester will check:

- whether the following user interface components exist with their respective subcomponents as stated in the analysis report.
  - File menu
  - Edit menu
  - View menu
  - Project menu
  - Debugger menu
  - Window menu
  - Help menu
  - Project window
  - Watch window
  - Memory Content window
  - Compiler Output window
  - Simulator GUI window
  - Simulator Configuration Dialog
  - Help Dialog
  - Programmer Dialog (a.k.a. Loader Module GUI)
- whether display windows and dialogs opened/closed, shown/hidden as instructed from menus and shortcuts.
- whether all windows and dialog can be opened/closed, moved, resized as expected normally from applications in the implementation platform, namely 32-bit Windows.
- whether shortcuts work as stated in the analysis report and shown in the menus, as expected normally from applications in the implementation platform.
- whether user interface behaves as expected in the test scenarios used in the functional requirements validation testing. (See next subsection) Particularly: whether changes in the program state displayed on the user screen and whether user inputs affect the program, promptly and correctly.

2.3.1.2 Functional Requirements Validation

Whether the program complies with the functional requirements as stated in the analysis report will be tested by running test scenarios.

Compiler Module

Compile single file (quick compile):

- start the program
- open an assembly/C source file
- select "Quick Build" from project menu
- check the output from the "Compiler Output" window
- go to the directory where the source file resides
- check whether the correct output files has actually been generated

Build Project:

- start the program
- open a project file that has multiple source files
- select "Build All" from the "Project" menu
- check the output from the "Compiler Output" window
- go to the directory where the project file resides
- check whether the correct output files has actually been generated
- edit a single source file
- select "Make" from the "Project" menu
- check whether the changed file is compiled
- check whether the changed file is the only one that is compiled

Project/File Manager Module

Add Files to/Remove Files From Project:

- start the program
- select "New Project" from "Project" menu
- select "Add Files to Project" from "Project" menu
- add several files to the Project
- try adding some of the files multiple times to the project
- try building the project. Check whether you are prompted to save the project after the changes.
- remove some of the files from the project

- try closing the project by selecting "Close Project" from "Project" menu. Check whether you are prompted to save the project after the changes.

Create / Modify / Save Files:

- start the program
- select "New" from "File" menu
- add a few line of text
- try closing the file. check whether you are prompted to save the project after the changes
- Open the file again. (e.g. by using ctrl+o short cut)
- try cutting/copying/pasting text
- try saving the file. No dialog should be displayed.

Simulator Module

- start the program
- set breakpoints
- adjust the input stimulus file
- start simulation
- step after each breakpoint
- stop simulation
- resume simulation
- simulation halts when reach end of file
- check whether the output is recorded correctly or not with the correct name

Loader Module

- start the program
- open connection
- select the program to be loaded to the board
- check whether loaded correctly or not
- if failed to load retry again
- if loaded completely close connection

2.3.1.3 Non-Functional Requirements

Although non-functional requirements affected our design profoundly, we will not be able to give their testing the importance it deserves, due to time limitations. Because issues with non-functional features (so called "ilities") arise in time only, as the program is being used, and the assessment is usually subjective, which makes it difficult to create concrete benchmarks -in contrast to, for example, writing a test program for unit testing-. We do not have time, alpha and beta testers, etc; we will use the following basic test procedures to help meet these requirements.

Performance / Resource Economy:

- Use the GNU Profiler (Windows port) to analyze program behavior with respect to time and space consumption.
- Make sure secondary storage space used by the total deployment of the program is in compliance to the goals stated in the analysis report.
- Make sure the program runs acceptably with a range of computers with various configurations.

Reliability / Stability / Robustness:

- Use a memory profiler (such as "mpatrol") to detect memory leaks, while running a simulation with the benchmark assembly programs used in unit testing.

Also:

- Start a simulation with one of the benchmark assembly programs used in unit testing.
- Leave it on overnight.
- Check the amount of page file usage using system tools. (Such as standard "Task Manager", or third party freeware "Process Explorer")
- Repeat with much shorter duration, e.g. 5 minutes.
- Observe if there is a significant difference in memory consumption that may have eluded memory profiler.

Usability:

- Check whether I/O bound operations affect the programs responsiveness.

2.3.2 Design Validation

2.4 Higher Order Testing

2.4.1 Performance Test:

To specify the performance boundaries we load the maximum executable hex file. Maximum load specifications must be known to protect the user from loader problems like paging problems. PIC16f877 doesn't need any complicated system requirements but there is a problem in the upper bounds of storing. If the hex file is bigger than 8kb, the system locks itself. This error occurs from the paging problems because it doesn't support data storage bigger than 8kb in the same file and divides the code into 8kb pages automatically. To protect this we have to divide the code into pages by own and then form the hex file .This will prevent paging problems.

2.4.2 Stress Test:

To test our program whether it works properly we have to include all modules of implementation and test it in the upper boundaries of performance limits. First we generate a code bigger than 8kb in size but divided into pages to protect paging problems and after generating the hex file we load it to the simulation module. We test the output of this by checking the leds, seven segment displays and lcd monitor whether there is a problem. After doing that we load the program into pic and analyze the differences of the simulation and the pic itself there should not be any problem or differences to prove the validity of our project.

2.4.3 Alpha and Beta Tests:

We do this project for ceng336 students so these tests are important. After releasing the alpha and beta versions of the program, for each we test the validity of our project and take this version to be tested by others. To test these versions we select 3 testers for both and try to take a feedback of these results. The testers' times are decided according to the time constraint and the requirement of development environment for alpha testing.

3 TESTING TOOL AND ENVIRONMENT

Our project will be developed by using Windows Operating system. We will use PC and CEng Card for testing. The testing tools and environments are Dev-C++, OpenGL library and our environments are Windows XP operating system.

4 REPRODUCIBILITY: FOR DEBUGGING

Debugging a failed test is the most important part of our task. After a failure is detected what will be done is explained in this section. When an error is detected during the unit testing, the developer that implemented the related module will try to debug the software and try to fix its error. However, if an error occurs during the integration testing, one of the developers will be responsible for fixing the error among which the combined modules are implemented by. For other testing procedures the errors will be fixed by one or more group members that are suitable for the job. During the debugging process, first the error will be recorded and reported to other members. Then related person or people that are going to be attributed to debugging will be decided. The problem with the software will be clarified and defined by the developers. By using the tools the code will be debugged. Testers will enter different and "interesting" inputs and outputs will be observed. It is impossible to fix all bugs. Therefore according to some specified inputs the program will be executed and the critical inputs and the reason of their usage -if clear and understood and possible- will be recorded and reported not to be confused during the execution of integrated modules. Although this is not a good practice in software development, since we have a shortage of time and developers we have to use some tricks. For a further release and advance one may improve our product and try to fix the unfixed bugs by tracing our bug report.

5 TESTING RESOURCES AND STAFFING

The duties of the staff for testing phase are as follows:

Simulator Unit Testing Fatih

Compiler Unit Testing Hayri

Project/File Unit Testing Adem

Loader Unit Testing Ulas

Blackbox Testing for First Release All Members

Blackbox Testing for Final Release All Members

Integration Testing All Members

System Validation All Members

6 TEST SCHEDULE

The following is the schedule for the testing plan that is presented in this report:

Test Plan Delivery: (Deadline) 06.05.2007

Unit Test and Integration Tests: (Deadline) 16.05.2007

Validation Tests: (Deadline) 22.05.2007

Performance and Stress Tests: (Start) 22.05.2007 - (Deadline) 25.05.2007

Beta and Alpha Tests: (Start) 01.06.2007 - (Deadline) 05.06.2007

Results (i.e. Bugs) Tracing and Correction: (Deadline) 10.06.2007