



Requirements Analysis Document

JEM ASM (A Multi-Architecture Assembly IDE)

Revision History:

Date:	Document Version:	Details:
Feb 15 2007	1.0	Initial Release
Feb 18 2007	2.0	3.3.2: Added text to explain the Use Cases 3.3.3: Added text to explain the Object Model

Preface:

This document addresses the requirements of an Integrated Development Environment that will support Assembly Language development and debugging for multiple instruction sets through an extendable design. The intended audience for this document is Client and JEM Development Team.

Target Audience:

Client, JEM Development Team

JEM Development Team:

Jack Blakely, Software Developer
Jen Couture, Software Quality Assurance
Erich Keane, Software Developer
Ed McCaffrey, Software Developer
Michael Tavares, Project Manager

1.0 General Goals

1.1 Purpose of the System

The purpose of JEM is to provide an easy to use IDE which allows Assembly developers to program and debug Assembly code for any target CPU architecture using a single development machine. The JEM IDE is designed to save time with its powerful debugging tools and reducing the need for learning multiple tool sets, as well as saving money in hardware debuggers for



embedded devices. The intuitive interface and debugging environment provides the ideal solution for learning assembly.

1.2 Scope of the System

JEM ASM focuses primarily on the development and debugging aspects of the software development cycle.

The IDE provided by JEM includes the developer tools that one would find in most modern IDEs. JEM supports an unlimited number of projects, and allows for each one to be developed for different target CPU architectures. JEM includes project and file management features, editing tools, a help system, full-featured plug-in system, and a powerful debugger.

JEM has the ability to emulate any CPU or microprocessor through the use of user-creatable Language and Architecture Definition files. The Language Definition system not only allows a user to add support for any CPU, but it also allows them to change the behavior of any existing command for testing purposes.

The debugger included with JEM provides a user with the ability to step through a program forwards and in reverse, set breakpoints, access the call stack, view the value of registers and variables, and edit those values during execution.

JEM does not include native or cross compilation. After using JEM to develop and debug an Assembly program, the user should use the assembler provided by the CPU manufacturer for optimal performance.

1.3 Objectives and Success Criteria of the Project

The main objective and feature of JEM is the ability to debug an Assembly program, as outlined in the scope, while it is running in an emulator which is dynamically created by JEM based on the Language and Architecture Definition files.

The ability for users to extend JEM support to any CPU architecture is more important than out-of-box support of any specific architecture. Therefore, the inclusion of multiple assembly instruction sets is not a priority; however, support for 16 bit Intel Architecture and at least one microprocessor is planned.



While responsive performance of the editor is an objective; emulation performance, in general, depends on the power of the host machine. Therefore, the performance of the debugger will depend on the speed of the developer's computer.

1.4 Definitions, Acronyms and Abbreviations

Assembly (ASM):

A low level programming language that is directly translatable into machine languages

Debugger:

A tool designed to execute a program in a special mode that allows the developer to step through the execution and monitor variables and memory to aid in correcting defects

Driver:

A program, sometimes written in Assembly, which is required to interface a piece of hardware with an Operating System or other pieces of hardware

Emulate:

A program that reproduces the behavior of a specific piece of hardware, and allows a user to execute a program designed for that hardware on the incompatible hardware that is running the emulator

GUI:

A Graphical User Interface is the visual environment which the user uses to interact with the computer

IDE:

An Integrated Development Environment is a program that combines common software development tools into one package

IPS:

The number of maximum Instructions per Second that can be executed by a processor

JEM:

JEM is the acronym composed of the original developers of the project. Those people being: Jack, Jen, Ed, Erich and Mike

Macro:

A macro is a symbol that represents a set of commands

Microcode:

An Assembly Instruction set that belongs to a specific microprocessor

Sandbox Debugging:



A Sandbox Debugger is a debugger that can execute a program within it, while preventing the program from having access to any resources outside of the debugger

1.5 References

Irvine, Kip R. *Assembly language for Intel-Bases Computers* Fourth Edition, Prentice Hall 1999 New Jersey

Kyprianou, Antonis. "WinAsm Studio." 14 Feb. 2007, 3:09 PM EST
<http://www.winasm.net/>

"RosAsm, the Bottom-Up Assembler for ReactOS" 14 Feb. 2007, 3:21 PM EST <http://betov.free.fr/RosAsm.html>

"X86 architecture." *Wikipedia, The Free Encyclopedia*. 6 Feb 2007, 15:20 UTC. Wikimedia Foundation, Inc. 8 Feb 2007
<http://en.wikipedia.org/w/index.php?title=X86_architecture&oldid=106062912>.

1.6 Overview

JEM is a replacement for hundreds of programs and hardware devices currently used for Assembly development and debugging. By combining the features of several pieces of software with the support for any Assembly instruction set, JEM becomes the best tool for all Assembly development.

JEM adds to the features currently seen in most Assembly development tools. JEM provides powerful file editing features, including entering and editing of text, full support of bookmarks and labels, code completion, and cut, copy, paste, find/replace and print. In addition to seamless editing tools, JEM allows a developer to add additional features with a powerful plug-in system.

The intuitive IDE of JEM is combined with the only debugger capable of creating emulators for any microprocessor. The highly integrated and feature-rich debugger provided by JEM can now be applied to all Assembly instruction sets, without the need for clumsy and expensive debugging hardware.

2.0 Current System

Currently there are different development and debugging tool sets for each microprocessor. There are no tools available that allow a developer to debug, on their development machine, Assembly



programs that are written for a different microprocessor than the one in that development machine.

There are two ways to debug a program in the current system. The first is software debugging on the machine that is running the program. This is the common solution seen with x86 Assembly development. The improvement that we offer in this scenario is a more powerful debugger that is more closely integrated with the editor, and a richer IDE with plug-in and code completion features. When the target device is an embedded machine that does not have the power or ability to run development tools, this is not a possible solution. The second way of debugging in the current system is with the use of a hardware debugger, as often seen in embedded devices.

Debugging on an embedded device is possible by connecting microprocessor-specific hardware to the device which will read the states of the registers and memory of your embedded device and return them to your IDE running on the development machine. After a change is made, the program must be saved and then transferred to the embedded device another time. This process is slow and the extra debugging hardware is expensive. JEM allows you to develop and debug Assembly programs written for any microprocessor architecture with only one development machine. There is no need for extra external debugging hardware, and no time is lost transferring programs to the device multiple times.

3.0 Proposed System

3.1 Functional Requirements

3.1.1 Programming Users

- 3.1.1.1 Create New Project
- 3.1.1.2 Open Existing Project
- 3.1.1.3 Save Project
- 3.1.1.4 Close Project
- 3.1.1.5 Install Plug-in
- 3.1.1.6 Remove Plug-in
- 3.1.1.7 Add New File to Project
- 3.1.1.8 Add Existing File to Project
- 3.1.1.9 Remove File From Project
- 3.1.1.10 Add Reference to Project
- 3.1.1.11 Remove Reference From Project
- 3.1.1.12 Open File
- 3.1.1.13 Close File
- 3.1.1.14 Enter Text
- 3.1.1.15 Edit Text
- 3.1.1.16 Paste Text
- 3.1.1.17 Undo Edit
- 3.1.1.18 Redo Edit



- 3.1.1.19 Open Command List
- 3.1.1.20 Select From Command List
- 3.1.1.21 Close Command List

3.1.2 Debugging Users

- 3.1.2.1 Pause Debugging
- 3.1.2.2 Step Forward
- 3.1.2.3 Step Into
- 3.1.2.4 View Variable List/Values
- 3.1.2.5 Edit Variable Values
- 3.1.2.6 View Call Stack
- 3.1.2.7 View Call Stack Function Call
- 3.1.2.8 Stop Debugging

3.1.3 All Users

- 3.1.3.1 Open Help
- 3.1.3.2 View About/Credits
- 3.1.3.3 Set Breakpoint
- 3.1.3.4 Remove Breakpoint
- 3.1.3.5 Start Debugging
- 3.1.3.6 Copy Text
- 3.1.3.7 Select All
- 3.1.3.8 Find/Replace
- 3.1.3.9 Set Bookmark
- 3.1.3.10 Remove Bookmark
- 3.1.3.11 Jump to Bookmark
- 3.1.3.12 Jump to Function
- 3.1.3.13 Jump to Label
- 3.1.3.14

3.2 Nonfunctional Requirements

3.2.1 Usability

JEM's IDE will be familiar to anyone with experience in developing with IDEs. There is also a help system to improve usability for less experienced users.

3.2.2 Reliability

The Architecture and Language Definition Files ensures that JEM always interprets and executes the Assembly instructions in the intended manner.

3.2.3 Performance

The management and editing features of JEM will be quick and responsive. Debugging relies on emulation, where the performance is heavily based on the speed of the emulating machine.

3.2.4 Supportability

An error-logging and reporting system provides a powerful supportability platform.

3.2.5 Implementation



JEM is a Windows application programmed in C# and can be run on any computer that runs Windows and has the .NET runtime environment installed.

3.2.6 Interface

The interface is similar to the standard IDE interfaces seen in other IDEs. A main text entry window will be supplemented with customizable windows to supply access to the other features outlined in this document.

3.2.7 Packaging

JEM will be available as a download over the Internet, as well as being offered on CD.

3.2.8 Legal

JEM is currently under plans to be released as a free product for non-commercial use.

3.3 System Models

3.3.1 Scenarios

Tom opens the JEM ASM IDE. Tom creates a new project and names it 'Mike'. Tom adds a reference to an external file for his project he just created. He types in some code for his 'Mike' program. Within the body of the code he creates a label. As he is typing in his command the code completion box appears and shows suggestions. Tom selects from one of the selections from the suggestion box. The code is completed for him. Tom changes the architecture associated with the project 'Mike'. He saves the project and exits the program.

Johan opens the application via the operating system's interface. He selects "Open Project" from the file menu. He navigates the project selection window, and selects "Open". He chooses a project to open in the new "Open Project" dialogue window. Johan sets a breakpoint by clicking on the line-number along the left side of the file edit window at the desired line.

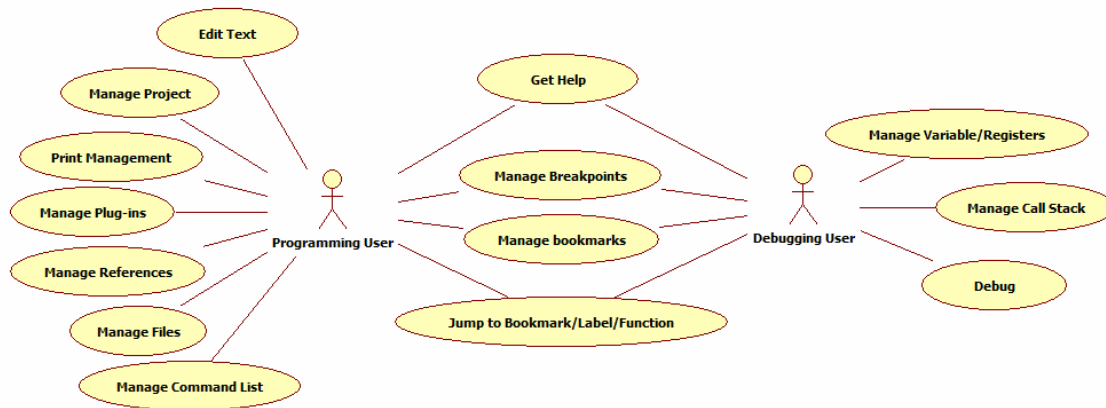
Next, he clicks the "Debug" arrow in the application toolbar. The debugger begins and automatically pauses the execution of the Assembly program at the previously set breakpoint. He selects "Step over", until he gets to a function call. Then, he clicks the "Step into" button on the Debug menu. The application executes the function, and changes the current highlighted line to the first inside the called function. Johan then opens the "Variable/Register View" Window, and checks on the values of his variables at this point in the execution.



Johan is finished debugging his application and selects the “Stop Debugging” button. He presses ctrl-S to save the project, and then closes the application.

3.3.2 Use Case Model

The two actors are used to show the separation between editing actions and debugging actions. There is no user management system, and users only interact with the system rather than exist as an entity within it.

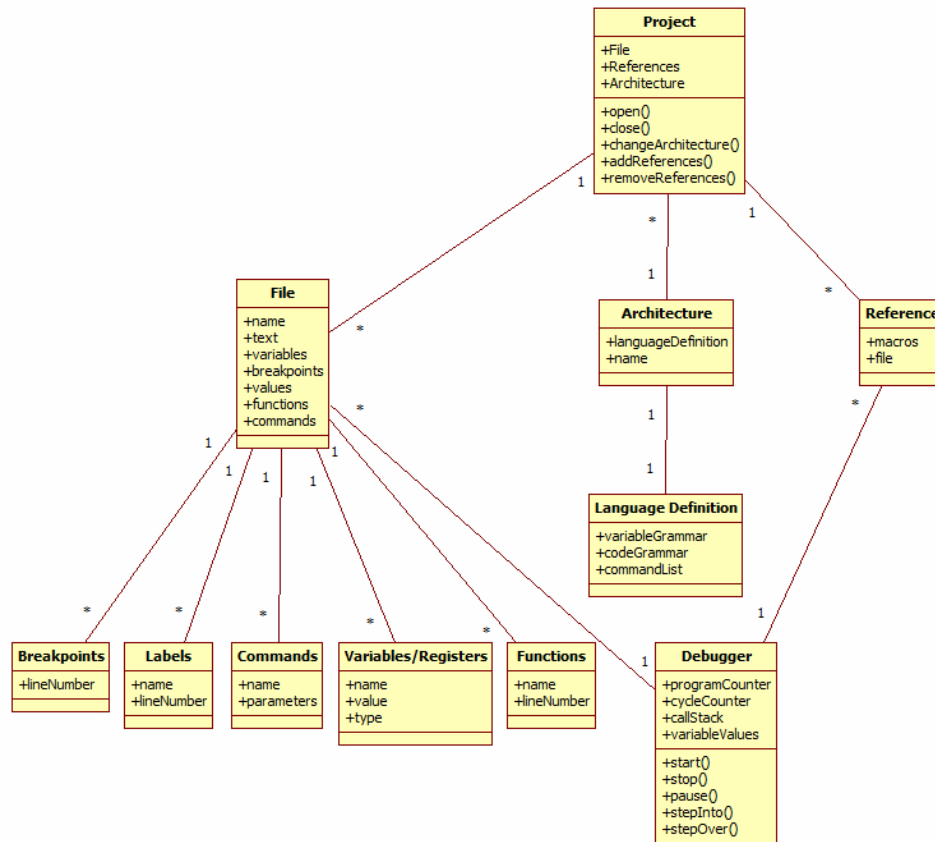


3.3.3 Object Model

This class model shows the objects that directly interface with the users. Every feature is made available after a project is opened or created. A project is a collection of files and settings that correspond with the program being developed. A reference is a macro, as defined in section 1.4, or a file. A file is an entity that can accept input from users and is used in the compilation and execution of a program. Files contain different types of input, which correspond to features of the Assembly language such as labels, commands, and functions, as well as entities to support debugging, such as breakpoints and information on the values of variables and registers. The Debugger class uses the Reference class to know which files exist in the current project, and it uses the File class to parse the code and create and run the program in debug mode. JEM supports all CPU architectures, and the Architecture class defines the target processor for the project. The Language Definition class defines what each instruction for that specific Assembly language



does and is used to tell the debugger how to handle each command.

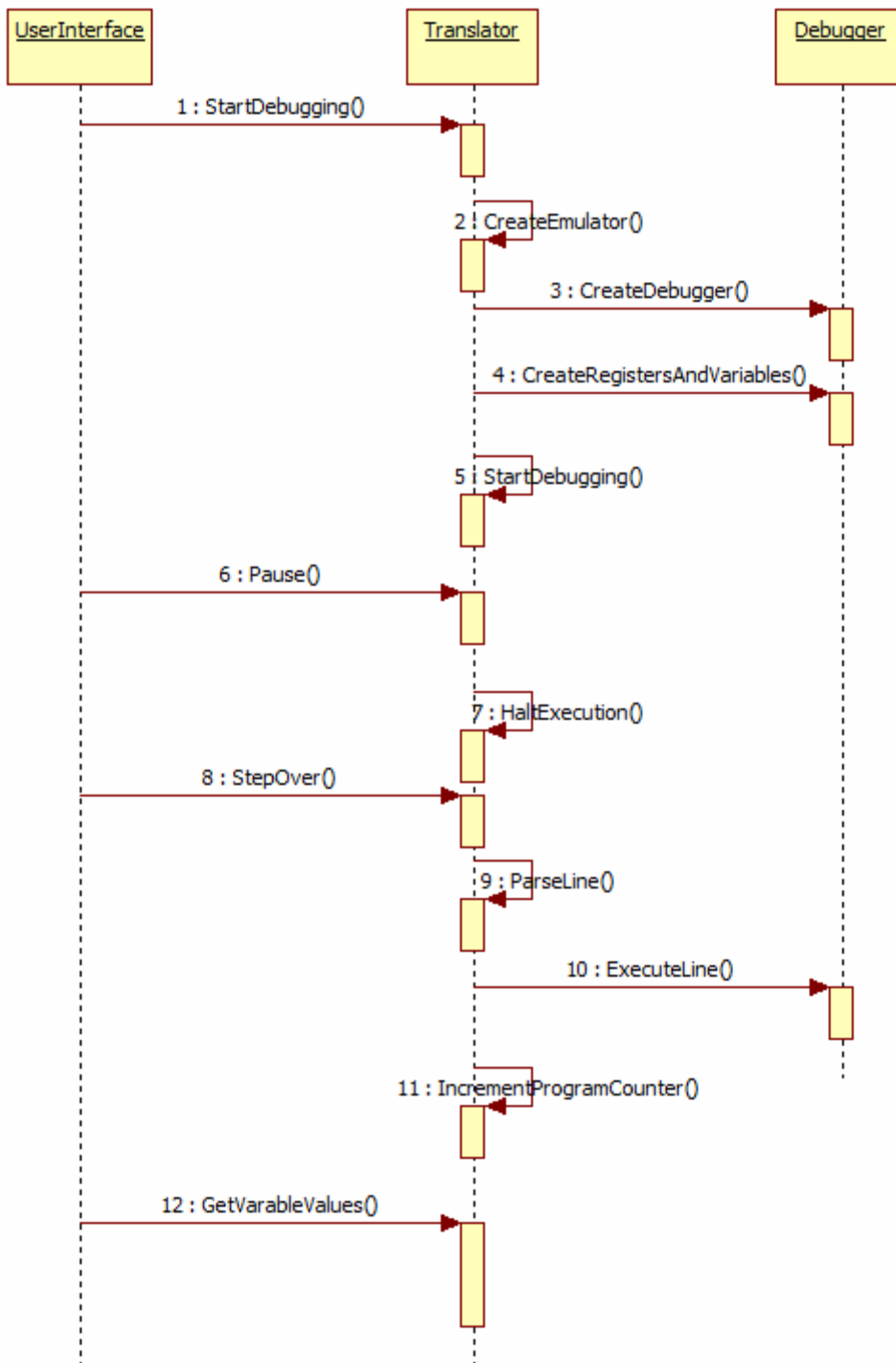




3.3.4 Sequence Diagrams

3.3.4.1 Step Over

The User Interface calls `StartDebugging()` which tells the Translator that the user wishes to begin debugging. The Translator invokes `CreateEmulator()`, which uses the Architecture and Language definition files to send the debugger the necessary information to create the emulator for the CPU architecture that the project uses. The Translator gives the translated information to the debugger via the `CreateDebugger()` method, and tells the debugger to setup the registers and variables according to the Architecture specification with `CreateRegistersandVariables()`. The Translator invokes `StartDebugging()` and uses the Architecture and Language definition files to translate the code in the files into instructions that the debugger understands, one instruction at a time. In this scenario, before any instruction is sent to the debugger the user pauses the execution by pressing the pause button in the UI. The UI calls `Pause()` and the Translator invokes `Halt()` which stops the translation and execution of instructions. The user wishes to resume execution of the program by manually stepping through the code. The user presses the Step Over button, which executes the next line of code without entering any functions. This action causes the UI to call `StepOver()` and the Translator invokes `ParseLine()` to translate the code into instructions the debugger understands, and then calls `ExecuteLine()` to execute the translated instruction. After the line has been executed, the Translator invokes `IncrementProgramCounter()` which is used to hold the address of the next instruction to be translated and executed. After a line of code has been executed, the values of variables and registers may change, hence the UI calls `GetVariableValues()` to update and display these values.

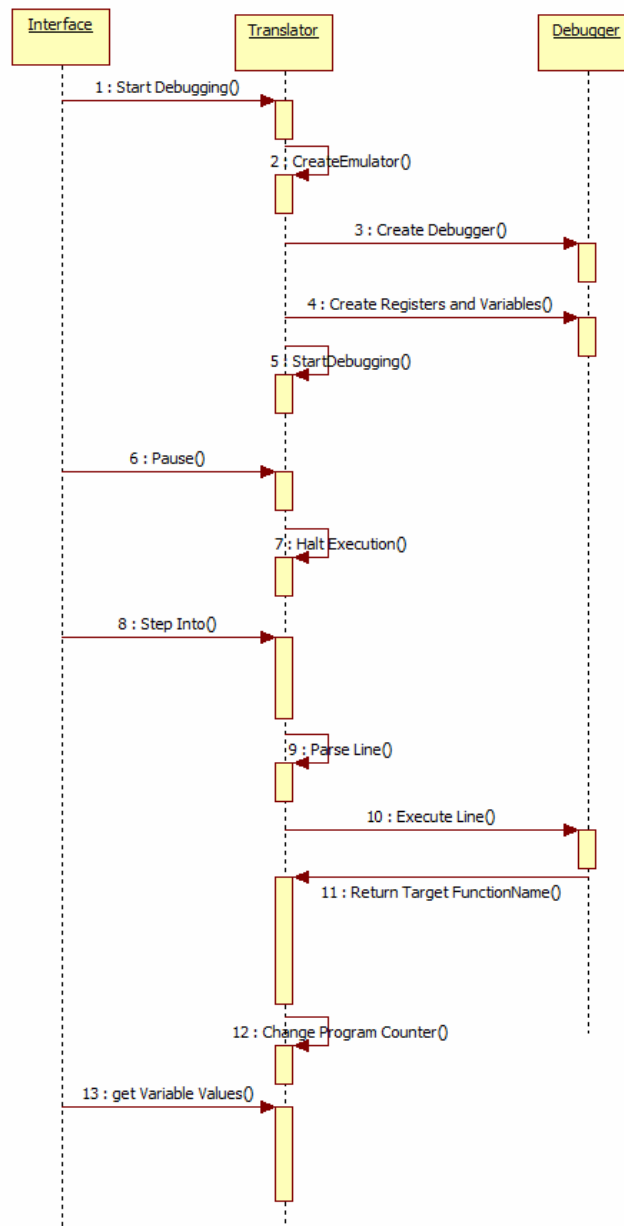




3.3.4.2 Step Into

The Step Into scenario is similar to the Step Over scenario, except it will enter functions instead of skipping them. The only difference in function calls are ReturnTargetFunctionName() and ChangeProgramCounter().

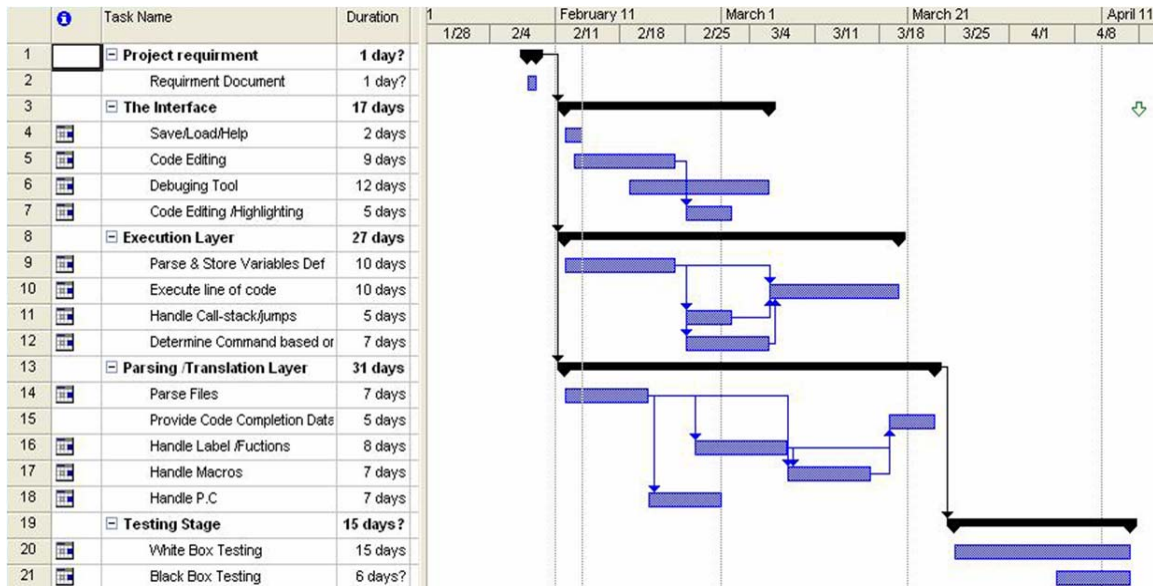
ReturnTargetFunctionName() is used when the function being executed changes due to a function call. ChangeProgramCounter() updates the Program Counted to reflect that change.





Costs, Benefits, and Scheduling

3.4 Gantt Chart



3.5 Estimated Costs

The project will take an estimated 368 person-hours to design, develop, test, and deploy. With a pay rate of \$45 per hour, the costs for labor is \$16,560. Additional costs of development machines, a source control server and software, office space, and employee benefits raise the estimated costs to a total of \$35,000.

3.6 Estimated Benefits

3.6.1 Tangible Benefits

JEM will reduce the cost of Assembly development to a fraction of the current costs. The streamlined IDE interface along with plug-in support, code completion, and preventing developers from spending time learning multiple tool sets will reduce development time by an estimated 25%. The powerful debugger will reduce development time by an estimated 50%. JEM's debugger also reduces the need for external hardware when developing for embedded systems, the costs of which must be calculated on a per-project basis.

At an hourly pay rate of \$45 per hour, JEM will reduce the costs by \$16.88 per hour by increasing productivity. With three developers working 40 hours per week, this adds up to \$8,102.40 saved per month.



3.6.2 Intangible Benefits

In addition to saving development time and costs, JEM will also decrease the frustration of developing in low level Assembly Language. A less stressful work environment will lead to a lower developer turnover rate. This can lead to additional developer efficiency.

By reducing the complexity of developing and debugging Assembly code, a developer will create code containing fewer defects. Higher quality code will increase customer approval and will lead to more business.

3.7 Payback Period

With an estimated total cost of \$35,000 and a benefit of \$8,102.40 per month, the payback period is just under 4.5 months. At 5 months, the company will have seen a profit of \$5,512, with a profit of \$8,102.40 each month thereafter.