Middle East Technical University
Department of Computer Engineering

CENG 491

Computer Engineering Design I

2006-2007

SimSys Corporation

Final Design Report

PIDE

Emulator and Development Environment for

CEng Embedded System Card

18.01.2007

Table of Contents

1. Introduction

1.1 Purpose of the Document

This document is prepared to supply the final design of the PIDE Project.

This report should be considered as final outcome of the design process. The work done and results are included in this document in a formal way. Since design process consists of modeling the system, the report contains diagrams and models of the current system. All the diagrams, design issues and models are meant to be final.

The report consists of three parts. In the "Modeling of the System" part, static and dynamic components of the system are represented. In the System and Project Specifications part, standards related to project implementation and various system components are introduced. In the Testing the System part, issues related to the final product testing are addressed and methods to be used are proposed.

1.2. Project Description

As the technology evolves, the embedded systems start to find wide area of usage. In most of the devices that people use daily, there exists a core logic which is mostly an embedded microcontroller or microprocessor with some external storage. Besides, those integrated devices also let the implementation and testing of various new controller ideas very easily. This popularity of embedded systems is a little overshadowed by the difficulty in developing embedded software due to the lack of a well fitted development environment and pre-testing it on a special independent system prepared just for testing purposes.

An example to the above discussion exists for the CEng336 Embedded Systems course. Among the course contents, development of embedded software and testing on a test board is of primary importance. However, obviously a standalone testing environment that will simulate exactly the same features with high accuracy would greatly simplify the testing procedure.

As a solution to the problem stated above, SimSys Corporation will develop an emulator and development environment, called PIDE (**P**IC **I**ntegrated

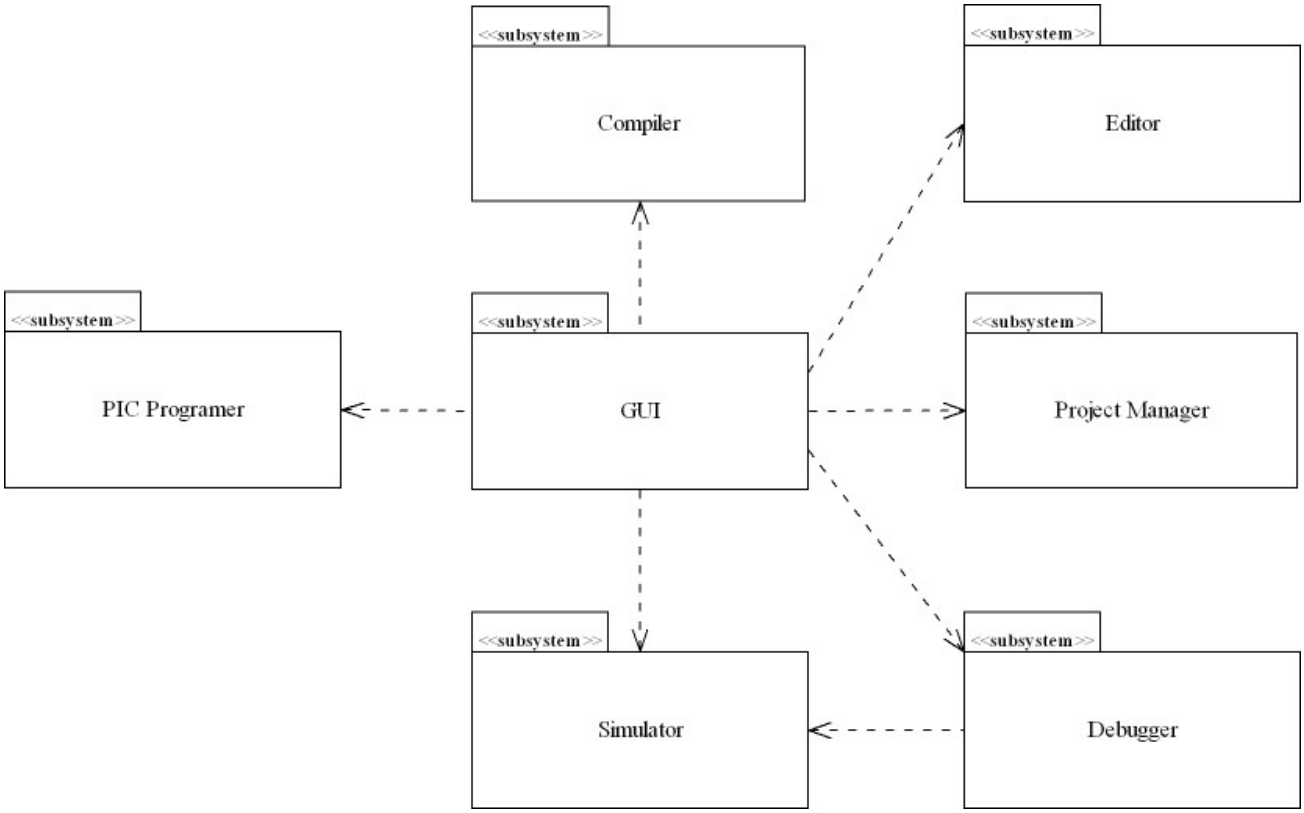
Development **E**nvironment), for the card used in Ceng336 Embedded Systems course. Considering such a development and simulation environment, the system will fully support the CENG336 board, i.e. support 16F877 PIC microcontroller and the other components on the board, communicate through various interface standards such as parallel, serial or USB and accommodate some display interfaces such as LCD or LED driving structures. Users will have the chance of compiling their programs and they can test and debug it on the virtual card emulated by the software. The virtual card will look the same as the CENG336 board.

PIDE is a real-time interactive and event-driven program. Among all of the interactions, the ones with the simulator is very important since the system should simulate the CENG336 Board with full functionality and give very similar responses to the user. Of course, it is not possible to give the same responses and simulate real-time behavior in a virtual environment because of reasons such as difference between the computational power of the PIC and the current computers, loss of data due to representing real time data, etc. But with careful estimations and assumptions, behavior of the board can be mimicked.

For such a development and simulation environment design project, the implementation areas are unlimited just as the fact that the implementation areas of the embedded systems are unlimited. As a result, such a system, which will simplify the development and testing process, will find great interest from the embedded systems developers. Together with the Ceng336 Card, this software will be useful for computer engineers, electrical engineers, high school students and everyone interested in PIC programming.

2. System Architecture

PIDE system is composed of several subsystems. The logical subsystem view of these components is represented below. The communication between the user and the system and among other components is managed by the GUI. GUI acts as the core of the system and each subsystem provides an interface to the core to make communication among each other. Other than the basic dependencies between subsystems and GUI, debugger depends on simulator. Debugger simulates the board using ExecutionController class which is inherited from Simulator class.



The main strategy for implementing subsystems is to initiate a main thread for each subsystem, divide their tasks into processes, handle each process by a thread and manage sub-threads by the main subsystem threads. Choosing a multi-threaded system architecture is inevitable for systems that requires high processing power and have frequent interactivity like PIDE.

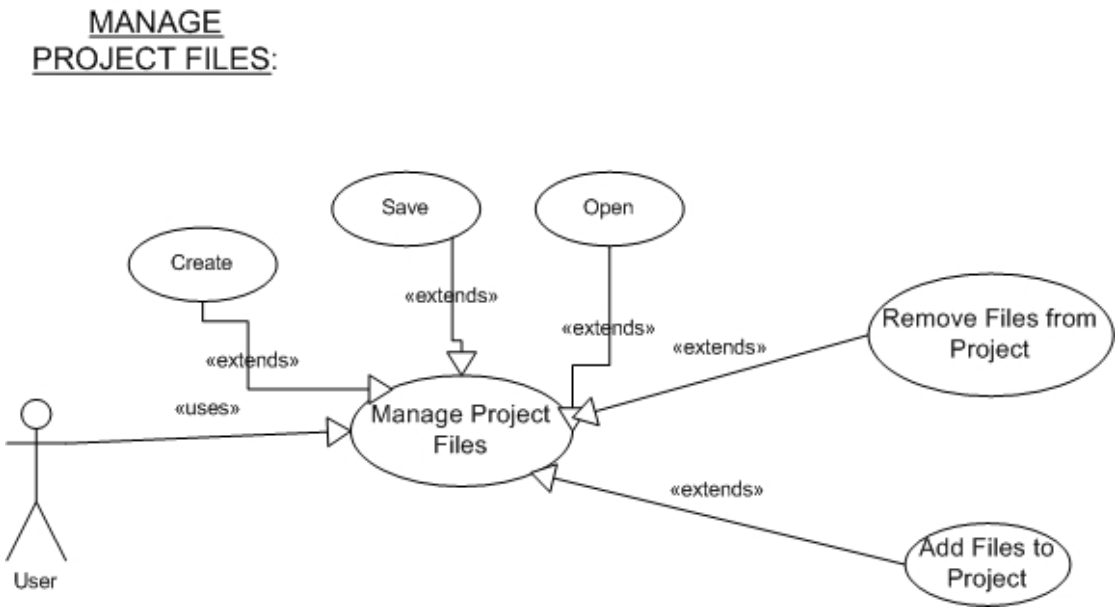
Since GUI is designed as the core of the program, it should be working all the time. GUI itself initiates several threads some of which are handled by Java classes and the others by PIDE classes. One thread of importance handled by PIDE classes is the BreakPointHandler thread. This thread loads breakpoint data from project file, passes the data to corresponding editor, compiler and debugger threads. Another important thread is for managing project-related operations and handled by Project Manager Subsystem. Editor module works as a separate thread and manages threads fired for each file opened in editor. The other modules initiate new threads as needed and their threads are killed after requested processing is finished. Having each subsystem handled by seperate a thread, the system can supply functionality without the user loosing the interaction with the system.

3. Modeling

3.1. Scenario Based Modeling

The use cases of the system describe the interaction between the system and the user from the user's point of view. This schematic is important to define the capabilities that are given to the user and his/her possible choices. There is no timing relationship existing in this diagram; however that information is given in the sequence diagrams, since these use cases are only to present the alternative paths that can be followed.

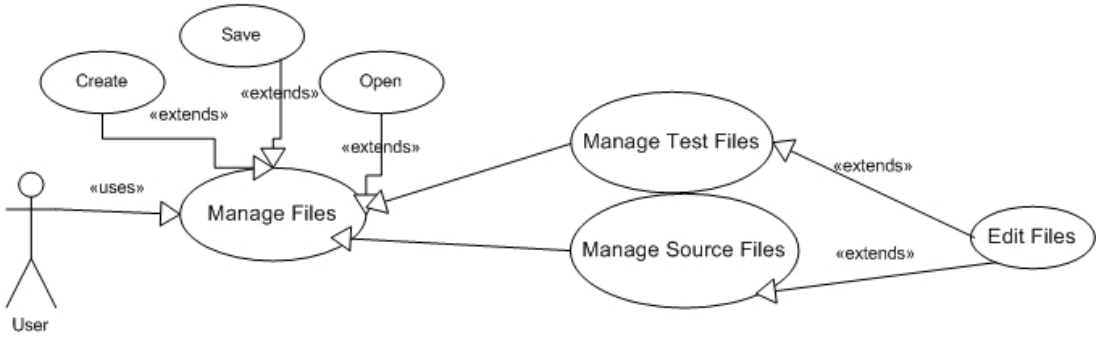
3.1.1. Manage Project Files



Managing a project is in fact handling of files within a project. Creation of new files, adding existing files to the project, removing files from the project are the possible tasks that can be performed in this use case. The files that are mentioned here may be of various types. The alternatives for file types are ASM++ source files, ASM source files and test bench files. Any change in the configuration of the projet is saved in corresponding project file.

3.1.2. Manage Files

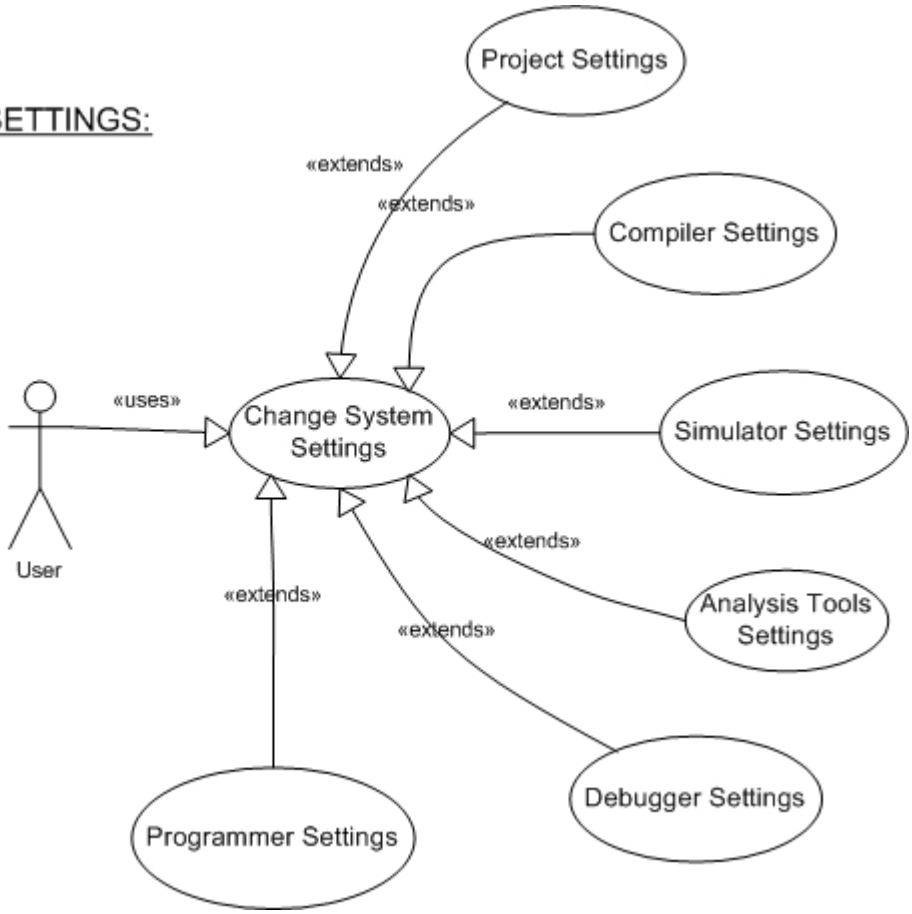
MANAGE FILES:



The user may select to manage the files using PIDE. Here, files may be created, saved, opened and edited. These files are the source files and test bench files. The source files are the ASM++ files or ASM files. The test bench file contains the input timing information for the peripherals.

3.1.3. Change Settings

CHANGE SETTINGS:



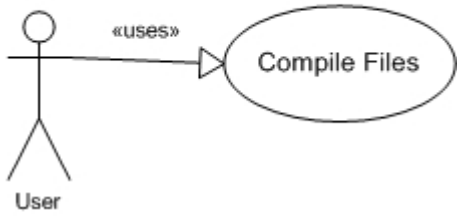
This use case defines the interaction of the user with the system to manage the settings of various internal modules of the software. Here, by means of graphical dialog windows, the user will be able to modify the system settings. This use case is in fact composed of a number of independent use cases. These are setting the project settings, compiler settings, simulator settings, debugger settings, analysis settings and finally the programmer settings. The first ones are self explanatory; however the last two require some elaboration.

Analysis settings are the specification of signals that are to be saved for later investigation. Here, some probes are inserted to the system, where the logic levels or voltages on those nodes are saved. Those saved waveform graphics can later be viewed via the analysis tool.

Programmer settings are about the programming interface of the board. Here, the parallel port selection can be performed and other choices about device programming can be made.

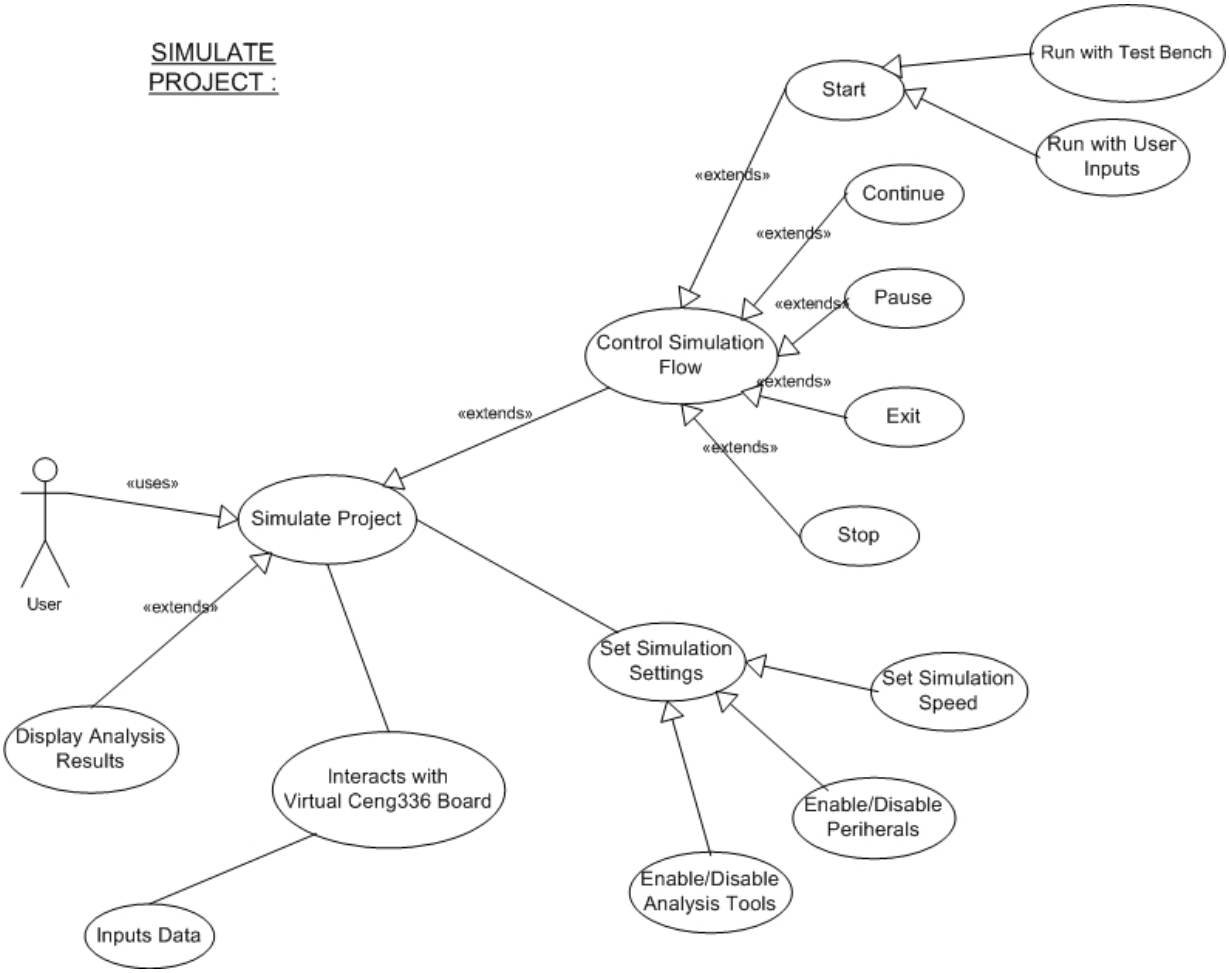
3.1.4. Compile Project

COMPILER:



The use case with the compile system is very straightforward. The user just requests a compile operation from the system. All syntax checking, parsing, linking and conversions are performed transparently to the user. The results are displayed in the output pane of the user interface.

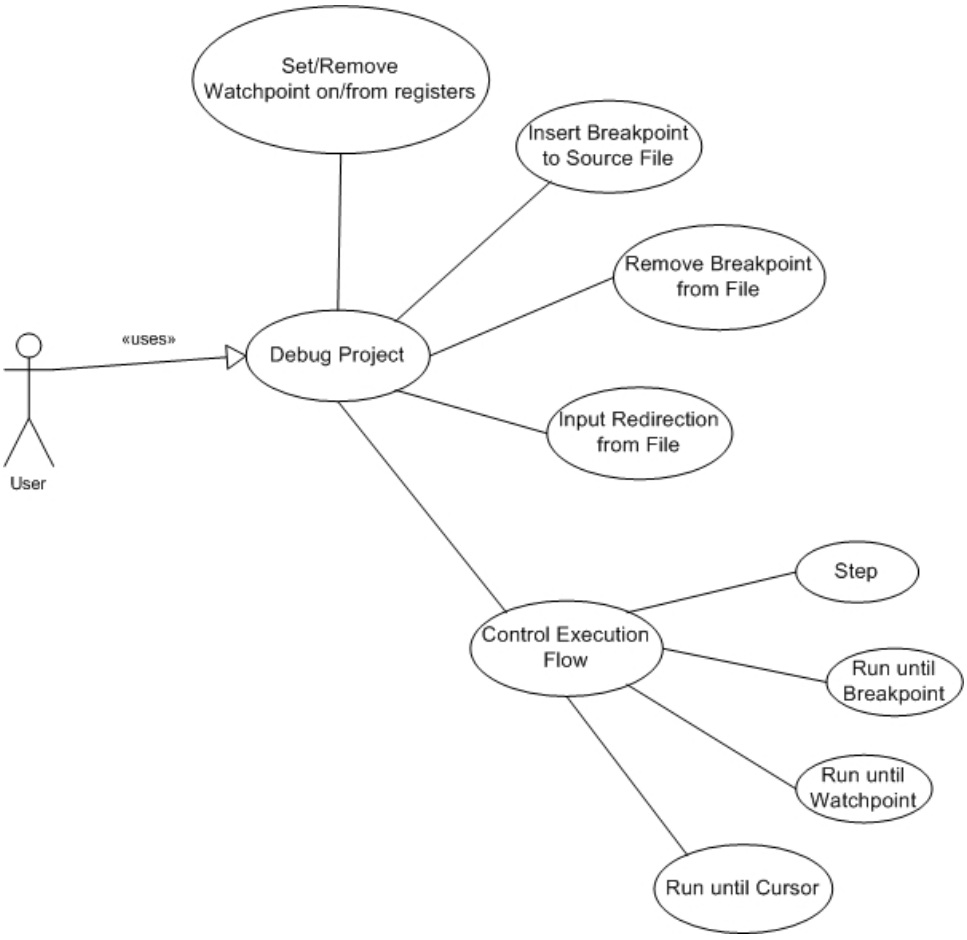
3.1.5. Simulate Project



In the simulation use case, the user will ask the system to run according to the specified inputs. The inputs may be provided by the user either in real time by means of the graphical user interface which is exactly the same as the layout of the board, or by some files that specifies some sequence of data to the input devices. These special files are called test bench files and have their special file format.

Simulation system has some special features. One of them is the enable/disable mechanism of the peripherals on the evaluation board. Another one is the selectable run speed. This feature will make the user much more comfortable in simulation of high frequency systems. For instance, in order to observe a signal toggling at 100 KHz, the system may be configured to run in 5us steps.

3.1.6. Debug Project

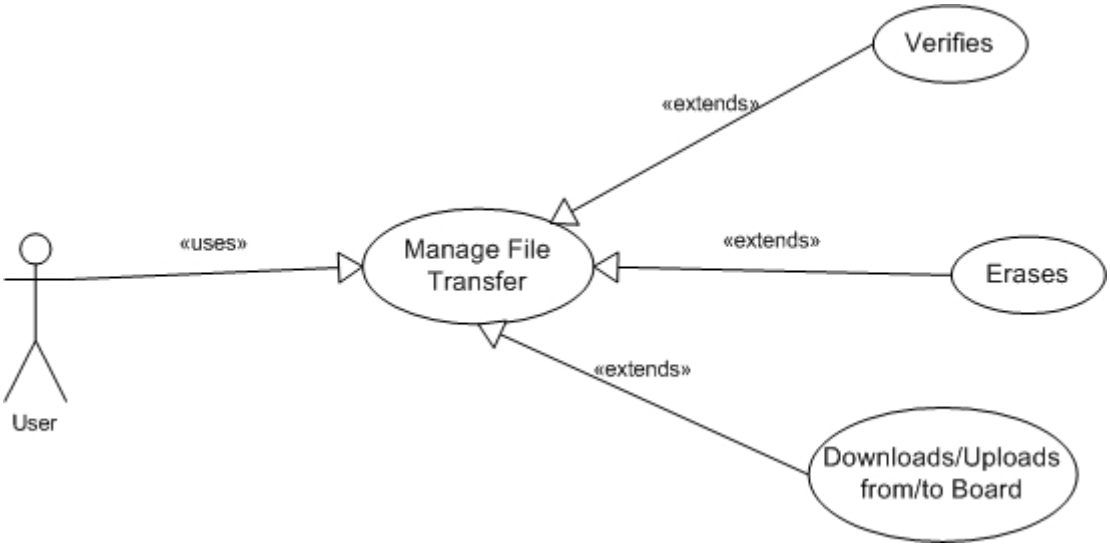


Debugging a project is to concentrate on the flow of the program on some specific parts of the source code. Debugging a project internally requires the project to be compiled and if current system is in not compiled state, then automatically the compile routine is invoked. Critical concepts for the debugger are the breakpoints and watch points.

Breakpoints are identifiers on some source code lines that state that the execution of the program will continue until that point and will halt there. The internal state of the system will be completely visible to the user, together with the contents of the registers. The execution flow will continue with some special events from the user such as a "step" command.

Watch points are identifiers attached to registers. These watch points are triggered when the value in the register is modified. The execution of the program halts at this point. Resuming is based on the same procedure as the one in breakpoints.

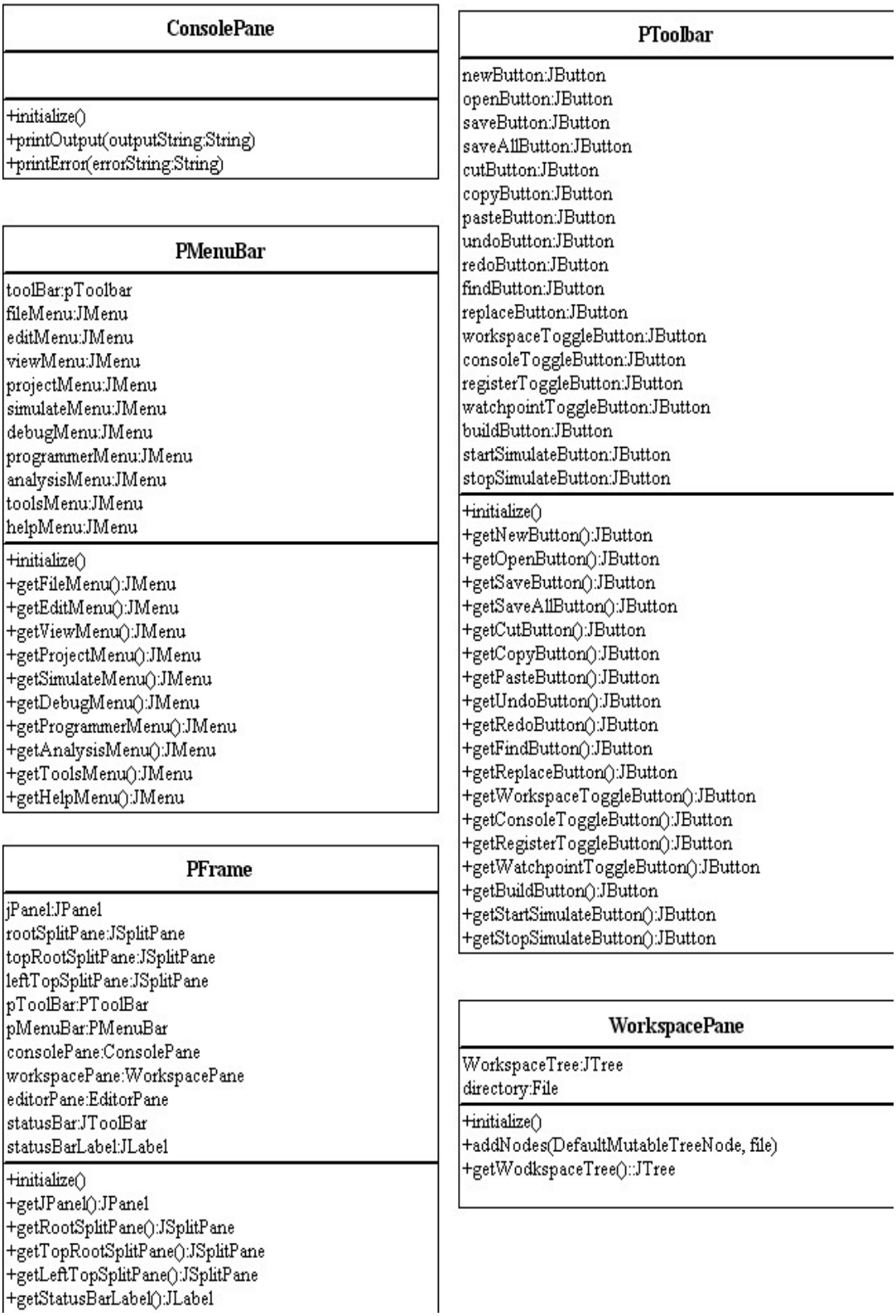
3.1.7. Manage File Transfer



Once the simulation is performed and the required results observed in the system, the user will upload the hex file to the microcontroller on the board to verify the operation physically. The user may also request to see the source of the program in currently residing in the microcontroller or may request a verification to check whether the uploaded program is consistent with the one in hand. The user may also want to clear the contents of the memory in the controller to be on the safe side and to start everything from scratch.

3.2. Object and Data Structure Modeling

3.2.1. Classes of *gui* Package



PFrame :: JFrame

This class defines the main outline of the PIDE GUI. It contains toolbar, menubar, statusbar as well as the console, debug, workspace and editor panes. It supplies methods to change the outline of the panes and other components.

	Attribute Name	Type		Description
Attributes	jPanel	JPanel		The main panel of GUI
	rootSplitPane	JSplitPane		JSplitPane instance that divides the main panel into top and bottom panels. Bottom split panel is the console panel.
	topRootSplitPane	JSplitPane		JSplitPane instance that divides the top panel of rootSplitPane into left and right panels. Right console panel is the debugger panel.
	leftTopSplitPane	JSplitPane		JSplitPane instance that divides the top panel of rootSplitPane into left and right panels. Left split panel is the workspace panel and right is the editor panel.
	pToolBar	PToolBar		The toolbar.
	pMenuBar	PMenuBar		The menubar.
	consolePane	ConsolePane		The console Pane.
	workspacePane	WorkspacePane		The workspace Pane.
	editorPane	editorPane		The editor Pane.
	statusBar	JToolBar		The status bar.
	statusBarLabel	JLabel		The label of the status bar.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	initialize()	void	void	Initializes the PFrame.
	getJPanel()	JPanel	void	Returns the jPanel item.
	getRootSplitPane ()	JSplitPane	void	Returns the rootSplitPane item.
	getTopRootSplitPane ()	JSplitPane	void	Returns the topRootSplitPane item.
	getLeftTopSplitPane ()	JSplitPane	void	Returns the leftTopSplitPane item.
	getStatusBarLabel ()	JLabel	void	Returns the statusBarLabel.

PMenuBar :: JMenuBar

This class holds all menu items and related methods of the PIDE menubar. Most functionality of the system can be carried using PIDE menubar.

	Attribute Name	Type	Description
Attributes	toolBar	PToolBar	Reference to he toolbar instance. Used to change view of the pToolBar.
	fileMenu	JMenu	The File menu item.
	editMenu	JMenu	The Edit menu item.
	viewMenu	JMenu	The View menu item.
	projectMenu	JMenu	The Project menu item.
	simulateMenu	JMenu	The Simulate menu item.
	debugMenu	JMenu	The Debug menu item.
	programmerMenu	JMenu	The Programmer menu item.
	analysisMenu	JMenu	The Analysis menu item.
	toolsMenu	JMenu	The Tools menu item.

	helpMenu	JMenu		The Help menu item.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	initialize()	void	void	Initializes the menu bar.
	getFileMenu()	JMenu	void	Returns the File menu item.
	getEditMenu()	JMenu	void	Returns the Edit menu item.
	getViewMenu()	JMenu	void	Returns the View menu item.
	getProjectMenu()	JMenu	void	Returns the Project menu item.
	getSimulateMenu()	JMenu	void	Returns the Simulate menu item.
	getDebugMenu()	JMenu	void	Returns the Debug menu item.
	getProgrammerMenu()	JMenu	void	Returns the Programmer menu item.
	getAnalysisMenu()	JMenu	void	Returns the Analysis menu item.
	getToolsMenu()	JMenu	void	Returns the Tools menu item.
	getHelpMenu()	JMenu	void	Returns the Help menu item.

PToolBar :: JToolBar

This class holds all toolbar buttons of PIDE. Some functionalities of PIDE are shortcutted via toolbar buttons.				
Attributes	**Attribute Name**	**Type**		**Description**
	newButton	JButton		The New button of the toolbar.
	openButton	JButton		The Open button of the toolbar.
	saveButton	JButton		The Save button of the toolbar.
	saveAllButton	JButton		The Save All button of the toolbar.
	cutButton	JButton		The Cut button of the toolbar.
	copyButton	JButton		The Copy button of the toolbar.
	pasteButton	JButton		The Paste button of the toolbar.
	undoButton	JButton		The Undo button of the toolbar.
	redoButton	JButton		The Redo button of the toolbar.
	findButton	JButton		The Find button of the toolbar.
	replaceButton	JButton		The Replace button of the toolbar.
	workspaceToggleButton	JButton		The Workspace Toggle button of the toolbar.
	consoleToggleButton	JButton		The Console Toggle button of the toolbar.
	registerToggleButton	JButton		The Register Toggle button of the toolbar.
	watchpointToggleButton	JButton		The Watchpoint Toggle button of the toolbar.
	buildButton	JButton		The Build button of the toolbar.
	startSimulateButton	JButton		The Start Simulate button of the toolbar.
	stopSimulateButton	JButton		The Stop Simulate button of the toolbar.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	initialize()	void	void	Initializes the tool bar.
	getNewButton()	JButton	void	Returns the New Button.
	getOpenButton()	JButton	void	Returns the Open Button.
	getSaveButton()	JButton	void	Returns the Save Button.

Methods	getSaveAllButton()	JButton	void	Returns the Save All Button.
	getCutButton()	JButton	void	Returns the Cut Button.
	getCopyButton()	JButton	void	Returns the Copy Button.
	getPasteButton()	JButton	void	Returns the Paste Button.
	getUndoButton()	JButton	void	Returns the Undo Button.
	getRedoButton()	JButton	void	Returns the Redo Button.
	getFindButton()	JButton	void	Returns the Find Button.
	getReplaceButton()	JButton	void	Returns the Replace Button.
	getWorkspaceToggleButton()	JButton	void	Returns the Workspace Toggle Button.
	getConsoleToggleButton()	JButton	void	Returns the Console Toggle Button.
	getRegisterToggleButton()	JButton	void	Returns the Register Toggle Button.
	getWatchpointToggleButton()	JButton	void	Returns the Watchpoint Toggle Button.
	getBuildButton()	JButton	void	Returns the Build Button.
	getStartSimulateButton()	JButton	void	Returns the Start Simulate Button.
	getStopSimulateButton()	JButton	void	Returns the Stop Simulate Button.

ConsolePane :: JPane

This class is responsible for the management of the console panel of PIDE. Console panel is used to output some system messages to the user.

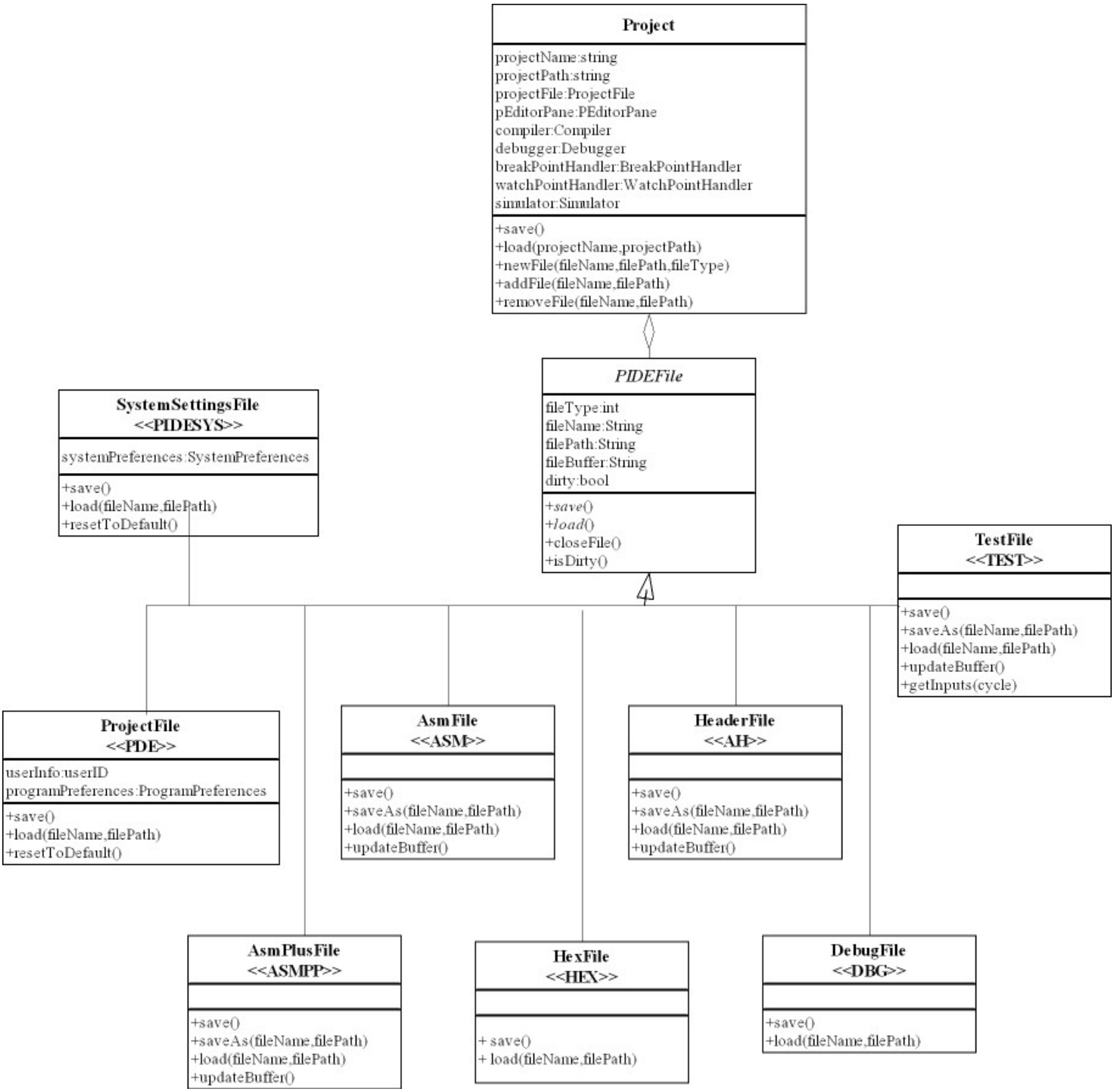
	Method Name	**Return**	**Arguments**	**Description**
Methods	initialize()	void	void	Initializes the Console Pane.
	printOutput()	void	string	Prints the output to the console.
	printError()	void	string	Prints the error message to the console.

WorkspacePane :: JPane

This class is responsible for the management of workspace panel of PIDE. Workspace panel shows the file and folder outline of the existing projects and supplies quick access to any file in a project.

	Attribute Name	**Type**		**Description**
Attributes	workspaceTree	JTree		The workspace tree structure.
	directory	File		The path name of the folder where project file is located.
	Method Name	**Return**	**Arguments**	**Description**
Methods	initialize()	void	void	Initializes the Workspace Pane.
	addNodes()	void	DeafultMutableTreeNode, file	Adds new nodes to the Workspace Tree.
	getWorkspaceTree()	JTree	void	Returns the Workspace Tree.

3.2.2. Classes of *projectManager* Package



Project

This class is responsible for the management of a project. It links the project files with editor, simulator, compiler and debugger modules. Any change in the project is performed by this class.				
Attributes	**Attribute Name**	**Type**		**Description**
	projectName	string		The name of the project.
	projectPath	string		The path of the project on the disk.
	projectFile	ProjectFile		The preferences of the project that are kept in file.
	pEditorPane	PEditorPane		Editor pane of the project.
	compiler	Compiler		The compiler module.
	debugger	Debugger		The debugger module.
	breakPointHandler	BreakPointHandler		The breakpoint handler.
	watchPointHandler	WatchPointHandler		The watchpoint handler.
	simulator	Simulator		The simulator module.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	save()	void	void	Saves the project.
	load()	void	projectName, projectPath	Loads the project.
	newFile()	void	fileType, fileName, filePath	Creates a new file and adds it to the project.
	addFile()	void	fileType, fileName, filePath	Adds an existing file to the project.
	removeFile()	void	fileType, fileName,	Removes a file from the project.

PIDEFile

This class is an abstract class. It is the base class which encapsulates basic functionalities of different types of files used by PIDE. PIDEFile classes represent the actual files stored on hard disc.				
Attributes	**Attribute Name**	**Type**		**Description**
	fileType	int		The type of the file: asm, hex, test, etc.
	fileName	string		The name of the file.
	filePath	string		The path of the file on the disk.
	fileBuffer	string		The buffer to hold the content of the file.
	dirty	bool		It will be true, when the file content has been changed after the last save.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	save()	void	void	Saves the file.
	load()	void	fileName, filePath	Loads the file.
	close()	void	void	Closes the file.
	isDirty()	bool	void	Returns the value of *dirty* variable.

SystemSettingsFile :: PIDEFile

This class is responsible for managing "pide.sys" file. SystemSettingsFile holds general information and preferences which affects overall program execution.			
Attributes	**Attribute Name**	**Type**	**Description**
	systemPreferences	SystemPreferences	The program preferences of the user.

	Method Name	Return	Arguments	Description
Methods	save()	void	void	Saves the file.
	load()	void	fileName, filePath	Loads the file.
	resetToDefault()	void	void	Resets the program preferences to default values.

ProjectFile :: PIDEFile

This class is responsible for managing files with ".pde" file extension. ProjectFile holds general information and preferences which affects only the corresponding project.				
Attributes	**Attribute Name**	**Type**		**Description**
	userInfo	userID		Information about the user of the project.
	projectPreferences	ProjectPreferences		The project preferences of the user.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	save()	void	void	Saves the file.
	load()	void	fileName, filePath	Loads the file.
	resetToDefault()	void	void	Resets the project preferences to default values.

ASMPlusFile :: PIDEFile

This class is responsible for managing files with ".asmpp" file extension. ASMPlusFile is the main source file of PIDE. The actual file is loaded into buffer and editing is performed on this buffer. Changes in the buffer is saved into actual buffer using one of the save methods.				
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	save()	void	void	Saves the file.
	saveAs()	void	fileName, filePath	Saves the file with a different name and/or to a different location.
	load()	void	fileName, filePath	Loads the file.
	updateBuffer ()	void	void	Updates the file with the current changes.

AsmFile :: PIDEFile

This class is responsible for managing files with ".asm" file extension. ASMFile is the basic source file of PIC microcontrollers. Any ASMPlusFile is first converted into this file type and then further process is performed on this file type during compilation. The actual file is loaded into buffer and editing is performed on this buffer. Changes in the buffer is saved into actual buffer using one of the save methods.				
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	save()	void	void	Saves the file.
	saveAs()	void	fileName, filePath	Saves the file with a different name and/or to a different location.
	load()	void	fileName, filePath	Loads the file.
	updateBuffer ()	void	void	Updates the file with the current changes.

HexFile :: PIDEFile

This class is responsible for managing files with ".hex" file extension. HexFile is the basic executable file of PIC microcontrollers. Any ASMFile is converted into this file type and then further process is performed on this file type. The actual file is loaded into buffer and editing is performed on this buffer. Changes in the buffer is saved into actual buffer using the save method.

	Method Name	Return	Arguments	Description
Methods	save()	void	void	Saves the file.
	load()	void	fileName, filePath	Loads the file.

HeaderFile :: PIDEFile

This class is responsible for managing files with ".ah" file extension. ASM header files are simply a reduced version of ASM source files. They include only procedure and macro definitions and can be included in the source files. The actual file is loaded into buffer and editing is performed on this buffer. Changes in the buffer is saved into actual buffer using one of the save methods.

	Method Name	Return	Arguments	Description
Methods	save()	void	void	Saves the file.
	saveAs()	void	fileName, filePath	Saves the file with a different name and/or to a different location.
	load()	void	fileName, filePath	Loads the file.
	updateBuffer ()	void	void	Updates the file with the current changes.

DebugFile :: PIDEFile

This class is responsible for managing files with ".dbg" file extension. Debug files store the information required for the debug process. They are outcome of compilation process. The actual file is loaded into buffer and editing is performed on this buffer. Changes in the buffer is saved into actual buffer using one of the save methods.

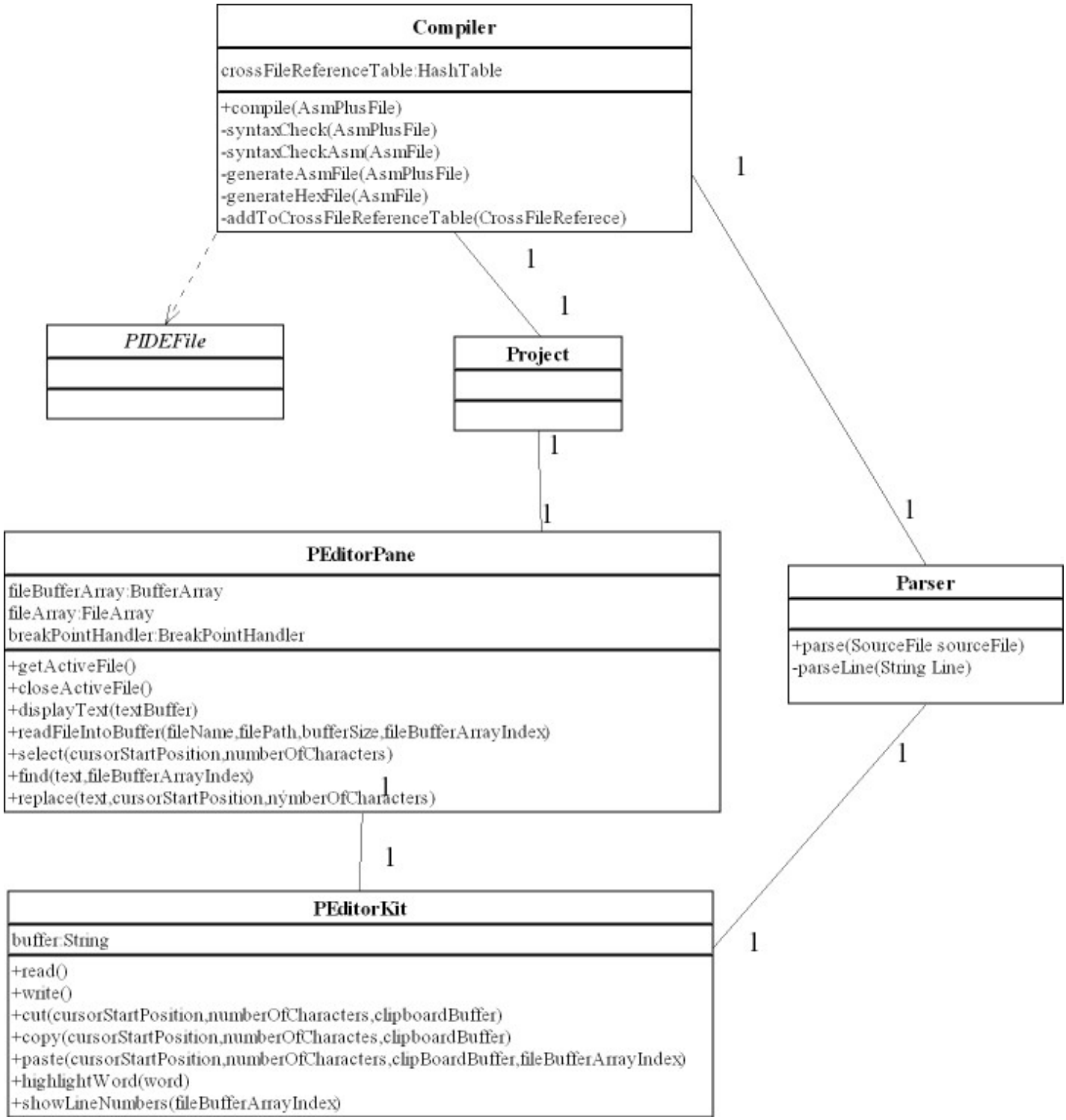
	Method Name	Return	Arguments	Description
Methods	save()	void	void	Saves the file.
	load()	void	fileName, filePath	Loads the file.

TestFile :: PIDEFile

This class is responsible for managing files with ".test" file extension. Test files input and timing data for automated simulations. They are user defined. The actual file is loaded into buffer and editing is performed on this buffer. Changes in the buffer is saved into actual buffer using one of the save methods.

	Method Name	Return	Arguments	Description
Methods	save()	void	void	Saves the file.
	saveAs()	void	fileName, filePath	Saves the file with a different name and/or to a different location.
	load()	void	fileName, filePath	Loads the file.
	updateBuffer()	void	void	Updates the file with the current changes.
	getInputs()	void	cycle	Gives the input values that should be applied at the given cycle.

3.2.3. Classes of *editor* and *compiler* Packages



PeditorPane :: JEditorPane

This class is responsible for editing files that user has edit permission. PeditorPane is actually a multi-file editor but it is designed so that it can be plugged into other systems. Additional functionality such as code highlighting is supplied with PEditorKit.

	Attribute Name	Type		Description
Attributes	fileBufferArray	BufferArray		The contents of the currently opened files.
	fileArray	fileArray		The files those are currently open.
	breakPointHandler	BreakPointHandler		The breakpoint handler.
	cursorPos	CursorPosition		Current position of the cursor
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	getActiveFile()	File	void	Returns the currently active file.
	closeActiveFile()	void	void	Closes the currently active file.
	displayText()	void	textBuffer	Displays the text in the buffer.
	readFileIntoBuf()	void	fileName, filePath, bufferSize, fileBufferArrayIndex	Reads the file into the specified buffer.
	select()	void	cursorStartPosition, numOfCharacters, fileBufferArrayIndex	Selects *numOfCharacters* characters starting from the *cursorStartPosition*.
Methods	find()	void	text, cursorStartPosition, fileBufferArrayIndex	Find *text* in the file.
	replace()	void	text, newText, cursorStartPosition, fileBufferArrayIndex	Find *text* in the file and replace with *newText*.

PeditorKit :: DefaultEditorKit

This class is responsible for supplying both basic and advanced functionalities required for an editor.

	Attribute Name	Type		Description
Attributes	buffer	string		The buffer of editor kit.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	read()	void	void	Reads a portion of the file into the buffer.
	write()	void	void	Writes the buffer into the file.
	cut()	void	cursorStartPosition, numOfCharacters, clipboardBuffer,	Puts the selected item into the clipboard buffer.
	copy()	void	cursorStartPosition, numOfCharacters, clipboardBuffer,	Copies the selected item into the clipboard buffer.
	paste()	void	cursorStartPosition, numberOfCharacters, clipBoardBuffer, fileBufferArrayIndex	Pastes the last item in the clipboard buffer.
	highlightWord()	void	word	Highlights the *word*.
	showLineNums()	void	fileBufferArrayIndex	Shows the line numbers.

Parser

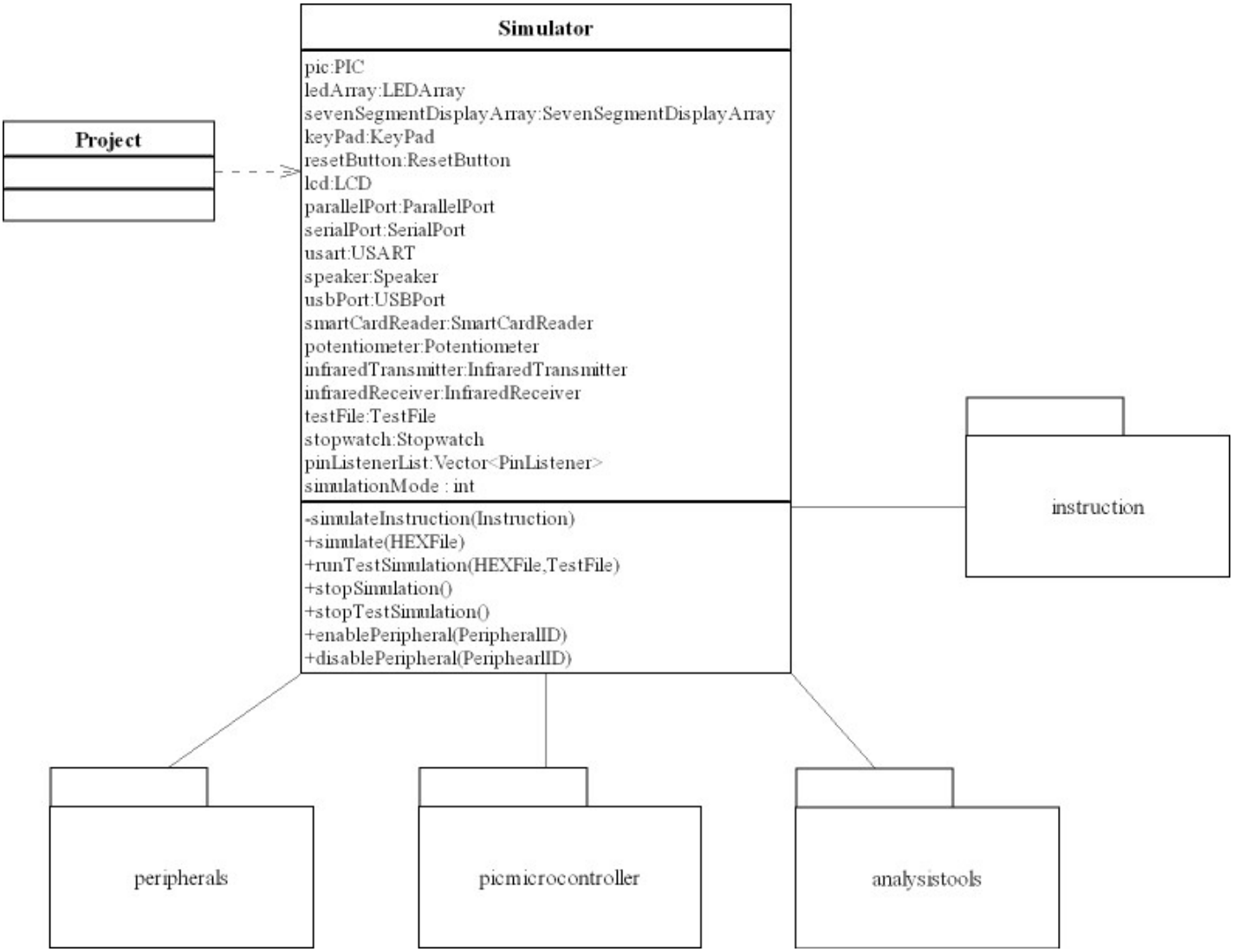
This class is responsible for parsing ASM and ASM++ source files and generating meaningful tokens for further evaluation during compilation.

	Method Name	Return	Arguments	Description
Methods	parse()	void	AsmPlusFile	Parses the file.
	parseLine()	TokenList	Line	Parses the given line.

Compiler

This class is responsible for compiling ASM++ source files. It supports both ASM and ASM++ files. An ASM++ file compilation generates an ASM file. The final product of the compile operation is a PIC-executable HEX file.				
Attributes	**Attribute Name**	**Type**		**Description**
	crossFileReference-Table	Hash Table		The mapping between the source file and the hex file.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	compile()	void	AsmPlusFile	Starts the compilation process.
	syntaxCheck()	void	AsmPlusFile	Checks the syntax of the AsmPlusFile.
	syntaxCheckAsm()	void	AsmFile	Checks the syntax of the AsmFile.
	generateAsm()	void	AsmPlusFile	Generates an AsmFile from the AsmPlusFile.
	generateHex()	void	AsmFile	Generates a HexFile from the AsmFile.
	addToCrossFile-ReferenceTable()	void	CrossFileReference	Adds the CrossFileReference entry to the CrossFileReferenceTable.

3.2.4. Classes of *simulator* Package



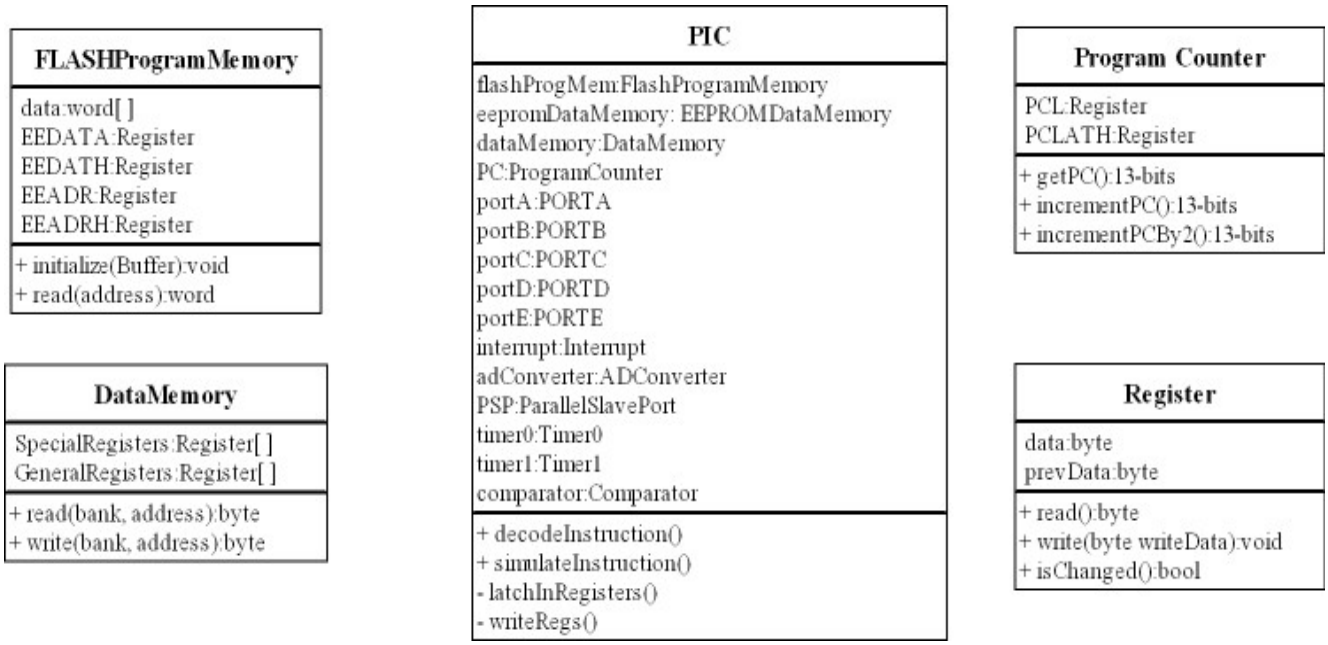
Simulator

This class manages the simulation operation and communicates virtual PIC with other peripherals and analysis tools. It provides methods to control simulation progress.

	Attribute Name	Type	Description
Attributes	pic	PIC	PIC microcontroller.
	ledArray	LEDArray	LED array on the board.
	sevenSegmentDisplayArray	SevenSegmentDisplayArray	7segment display array on the board.
	keyPad	KeyPad	Keypad on the board.
	resetButton	ResetButton	Reset button on the board.
	lcd	LCD	LCD display on the board.
	parallelPort	ParallelPort	Parallel port on the board.
	serialPort	SerialPort	Serial port on the board.
	usart	USART	USART module on the board.
	speaker	Speaker	Speaker on the board.
	usbPort	USBPort	USB port on the board.
	smartCardReader	SmartCardReader	Smart card reader on the board.
	potentiometer	Potentiometer	The analog input POT on the board.
	infraredTransmitter	InfraredTransmitter	Infrared-transmitter on the board.
	infraredReceiver	InfraredReceiver	Infrared-receiver on the board.
Attributes	testFile	TestFile	Test bench data for simulation.
	stopwatch	Stopwatch	Stopwatch to keep the time during simulation.
	pinListenerList	Vector<PinListener>	Pin listener to keep the logic values of the pins.
	simulationMode	int	The mode of the simulation.

	Method Name	Return	Arguments	Description
Methods	simulate()	void	HexFile	Makes the simulation.
	runTestSimulation()	void	HexFile, TestFile	Makes the test bench simulation.
	stopSimulation()	void	void	Stop the simulation.
	stopTestSimulation()	void	void	Stop the test bench simulation.
	enablePeripheral()	void	PeripheralID	Enables the peripheral in the simulation
	disablePeripheral()	void	PeripheralID	Disables the peripheral in the simulation

3.2.4.1. Classes of *picmicrocontroller* Package



PIC

This class simulates a PIC16F877 microcontroller. It simulates one instruction at a time and does necessary changes in its components.				
Attributes	**Attribute Name**	**Type**		**Description**
	flashProgMemory	FlashProgramMemory		Flash program memory
	eepromDataMemory	EEPROMDataMemory		EEPROM data memory
	dataMemory	DataMemory		Data memory
	pc	ProgramCounter		Program Counter
	portA	PORTA		PORT A of the PIC
	portB	PORTB		PORT B of the PIC
	portC	PORTC		PORT C of the PIC
	portD	PORTD		PORT D of the PIC
	portE	PORTE		PORT E of the PIC
	interrupt	Interrupt		Interrupt module of the PIC
	adConverter	ADConverter		Analog-to-Digital Converter
Attributes	psp	ParallelSlavePort		Parallel Slave Port
	timer0	Timer0		Timer 0 of the PIC
	timer1	Timer1		Timer 1 of the PIC
	comparator	Comparator		Comparator of the PIC
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	decodeInstruction()	void	void	Decodes the next instruction.
	simulateInstruction()	void	void	Simulates the next instruction.
	latchInRegs()	void	void	Latch in the register values before the execution of a step.
	writeRegs()	void	void	Write the updated values of the registers after a step.

Register

This class represents a simple register of the data memory of PIC16F877 microcontroller.				
Attributes	**Attribute Name**	**Type**		**Description**
	data	byte		The content of the register
	prevData	byte		Previous content of the register
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	read()	byte	void	Reads the data in the register.
	write()	void	byte	Writes the *byte* into the register.
	isChanged()	bool	void	Returns true if the content of the register has been changed, returns false otherwise.

FlashProgramMemory

This class represents the flash program memory of PIC16F877 microcontroller.				
Attributes	**Attribute Name**	**Type**		**Description**
	data	word[]		The content array of the memory (each element is 14 bits)
	EEDATA	Register		EEDATA register
	EEDATH	Register		EEDATH register
	EEADR	Register		EEADR register
	EEADRH	Register		EEADRH register
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	initialize()	void	Buffer	Initializes the memory.
	read()	14bit-data	Address	Read the data at the *Adress*.

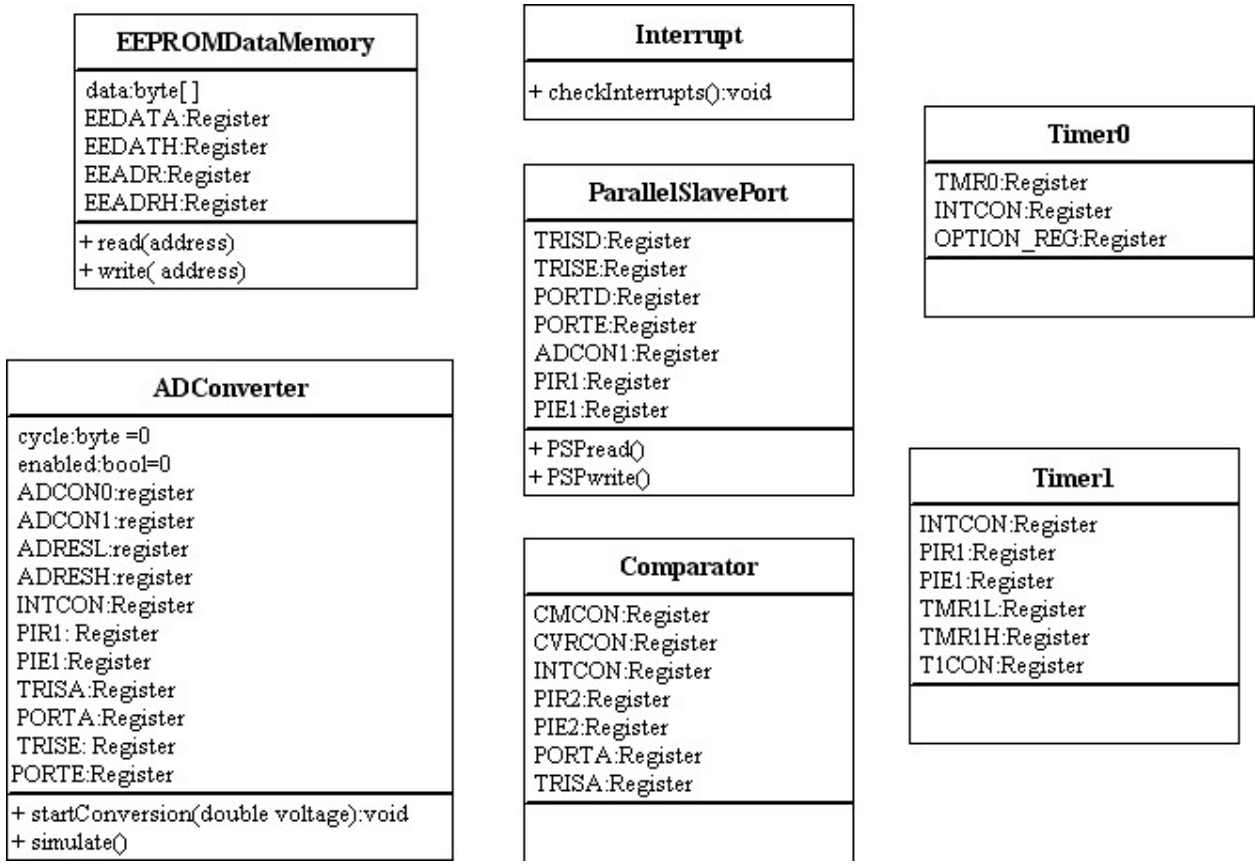
DataMemory

This class represents data memory of PIC16F877 microcontroller.				
Attributes	**Attribute Name**	**Type**		**Description**
	specialRegisters	Register[]		The special registers in Data Memory.
	generalRegisters	Register[]		The general registers in Data Memory.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	read()	byte	Bank, Address	Reads the byte at the *Adress* on *Bank*.
	write()	void	Bank, Address, byte	Writes the *byte* to the *Adress* on *Bank*.

ProgramCounter

This class encapsulates related data and methods for the program counter of PIC16F877 microcontroller.				
Attributes	**Attribute Name**	**Type**		**Description**
	PCL	Register		PCL Register in the PIC
	PCLATH	Register		PCLATH Register in the PIC (only 5 bits are meaningful)
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	get()	13bit	void	Gets the current value of the PC.

	increment()	13bit	void	Increments the value of the program counter.
	incrementBy2()	13bit	void	Increments the value of the program counter by 2.



EEPROMDataMemory

This class represents EEPROM data memory of PIC16F877 microcontroller.				
Attributes	**Attribute Name**	**Type**		**Description**
	data	byte[]		The content array of the memory (each element is 8 bits)
	EEDATA	Register		EEDATA register
	EEDATH	Register		EEDATH register
	EEADR	Register		EEADR register
	EEADRH	Register		EEADRH register
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	read()	byte	Address	Reads the *byte* at the *Adress*.
	write()	void	Address, byte	Writes the *byte* to the *Adress*.

ADConverter

This class represents the analog-to-digital converter of the PIC16F877 microcontroller.			
Attributes	**Attribute Name**	**Type**	**Description**
	cycle	byte	AD conversion cycle
	enabled	bool	If AD conversion is enabled

	ADCON0	Register		ADCON0 Register
	ADCON1	Register		ADCON1 Register
	ADRESL	Register		ADRESL Register
	ADRESH	Register		ADRESH Register
	INTCON	Register		INTCON Register
	PIR1	Register		PIR1 Register
	PIE1	Register		PIE1 Register
	PORTA	Register		Local copy of PORTA
	PORTE	Register		Local copy of PORTE
	TRISA	Register		Local copy of TRISA
	TRISE	Register		Local copy of TRISE
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	startConversion()	void	double	Starts the AD conversion of the given analog voltage.
	simulate()	void	void	Simulates the AD conversion.

Interrupt

This class handles the interrupt routines of PIC16F877 microcontroller.				
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	checkInterrupts()	Void	Void	Checks if there are interrupts.

ParallelSlavePort

This class represents the Parallel Slave Port of PIC16F877 microcontroller.				
Attributes	**Attribute Name**	**Type**		**Description**
	PORTD	Register		Local copy of PORTD
	PORTE	Register		Local copy of PORTE
	TRISD	Register		Local copy of TRISD
	TRISE	Register		Local copy of TRISE
	ADCON1	Register		Local copy of ADCON1
	PIR1	Register		Local copy of PIR1
	PIE1	Register		Local copy of PIE1
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	pspRead()	void	Void	Read the data.
	pspWrite()	void	Void	Write the data.

Timer0

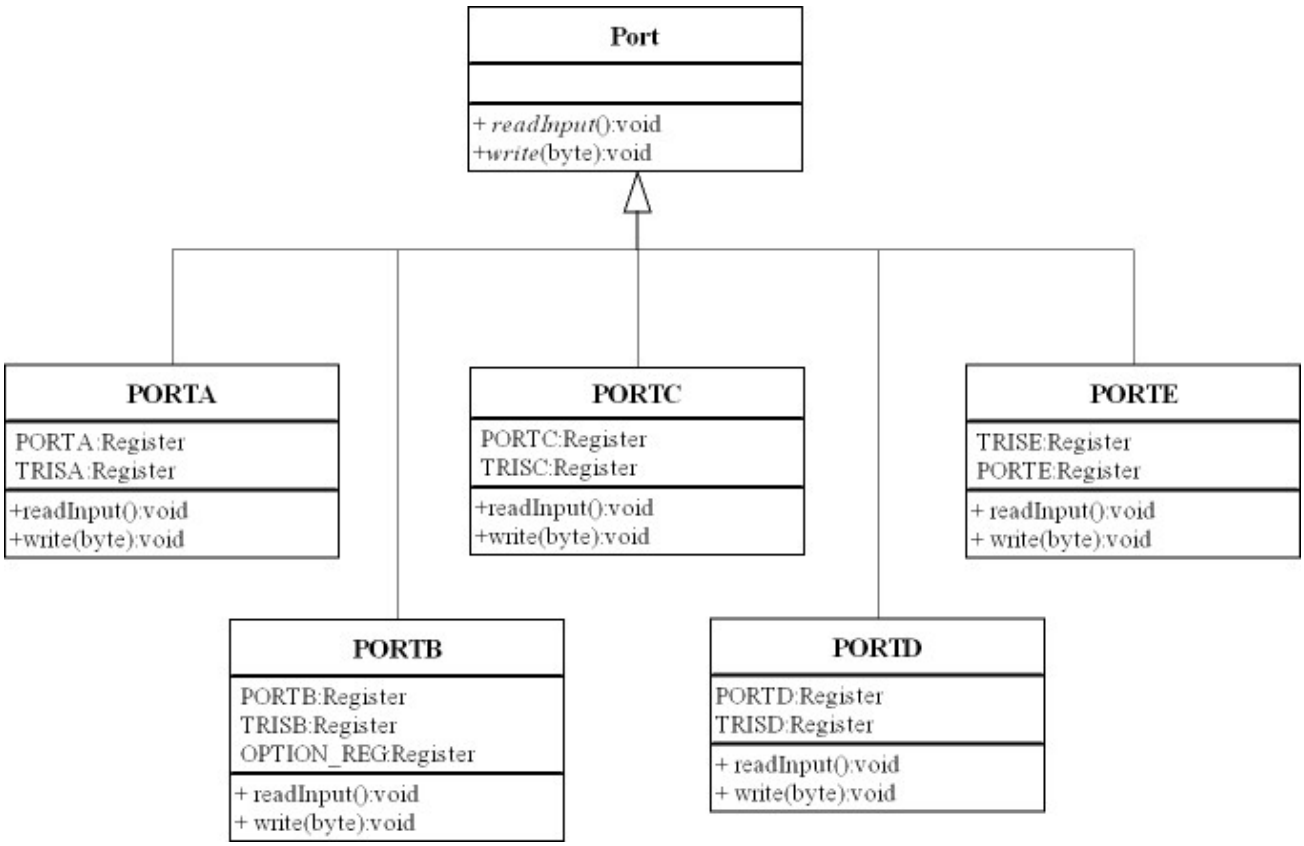
This class represents Timer0 of PIC16F877 microcontroller.				
Attributes	**Attribute Name**	**Type**		**Description**
	TIMER0	Register		Local copy of TIMER0
	INTCON	Register		Local copy of INTCON
	OPTION_REG	Register		Local copy of OPTION_REG

Timer1

This class represents Timer1 of PIC16F877 microcontroller.			
	Attribute Name	**Type**	**Description**
Attributes	INTCON	Register	Local copy of INTCON
	PIR1	Register	Local copy of PIR1
	PIE1	Register	Local copy of PIE1
	TMR1L	Register	Local copy of TMR1L
	TMR1H	Register	Local copy of TMR1H
	T1CON	Register	Local copy of T1CON

Comparator

This class represents the registers related with the compare mode of PIC16F877 microcontroller.			
	Attribute Name	**Type**	**Description**
Attributes	CMCON	Register	Local copy of CMCON
	CVRCON	Register	Local copy of CVRCON
	INTCON	Register	Local copy of INTCON
	PIR2	Register	Local copy of PIR2
	PIE2	Register	Local copy of PIE2
	PORTA	Register	Local copy of PORTA
	TRISA	Register	Local copy of TRISA



Port

This class represents the ports of PIC16F877 microcontroller. Specific ports are inherited from this class.				
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	readInput()	byte	Void	Reads the input data in the port.
	write()	void	Byte	Writes the data into the port.

PORTA :: Port

This class represents PortA.				
Attributes	**Attribute Name**	**Type**		**Description**
	PORTA	Register		The content of the Port register (only 6 bits are meaningful)
	TRISA	Register		The data direction Register
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	readInput()	byte	Void	Reads the input data in the port.
	write()	void	Byte	Writes the data into the port.

PORTB :: Port

This class represents PortB.				
Attributes	**Attribute Name**	**Type**		**Description**
	PORTB	Register		The content of the Port register
	TRISB	Register		The data direction Register
	OPTION_REG	Register		OPTION_REG Register
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	readInput()	byte	Void	Reads the input data in the port.
	write()	void	Byte	Writes the *byte* into the port.

PORTC :: Port

This class represents PortC.				
Attributes	**Attribute Name**	**Type**		**Description**
	PORTC	Register		The content of the Port register
	TRISC	Register		The data direction Register
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	readInput()	byte	Void	Reads the input data in the port.
	write()	void	Byte	Writes the *byte* into the port.

PORTD :: Port

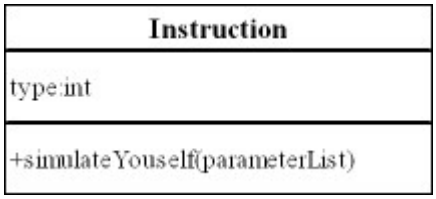
This class represents PortD.				
Attributes	**Attribute Name**	**Type**		**Description**
	PORTD	Register		The content of the Port register
	TRISD	Register		The data direction Register
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	readInput()	byte	Void	Reads the input data in the port.

	write()	void	Byte	Writes the *byte* into the port.

PORTE :: Port

This class represents PortE.				
Attributes	**Attribute Name**	**Type**		**Description**
	PORTE	Register		The content of the Port register
	TRISE	Register		The data direction Register
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	readInput()	byte	Void	Reads the input data in the port.
	write()	void	Byte	Writes the *byte* into the port.

3.2.4.2. Classes of *instructions* Package

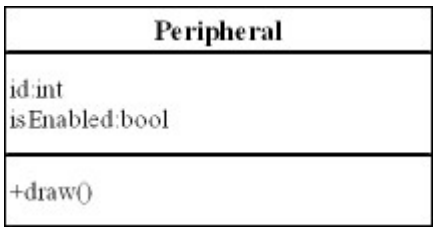


Instruction

This class is an abstract base class for the simulation of PIC mcu instructions.				
Attributes	**Attribute Name**	**Type**		**Description**
	type	int		The type of the instruction. (From 1 to 35).
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	simulateYourself()	void	parameterList	Simulates the instruction with the given parameters. Will be overwriten in the derived classes.

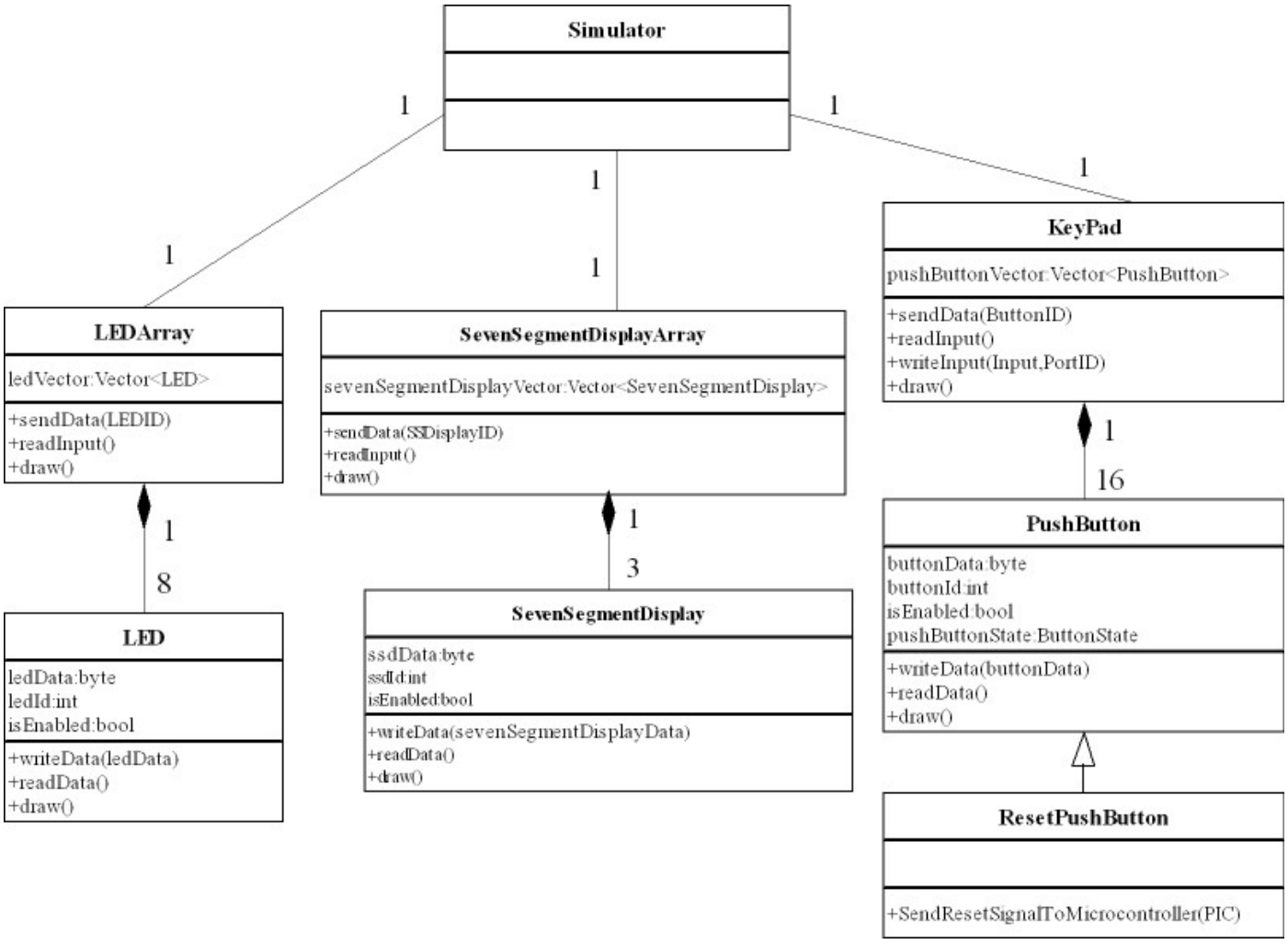
PIC microcontroller has 35 instructions and we have a class for each of these instructions, which are all derived from the class Instruction. The .hex file to be simulated will first be converted to a list of instructions, then *simulateYourself()* function of each element of the list will be called. Each instruction class is responsible for simulating itself, i.e. performing the required operation, updating the required registers, etc.

3.2.4.3. Classes of *peripherals* Package



Peripheral

This is a general class representing the peripherals on the CENG336 Board. Specific peripherals are inherited from this class.				
Attributes	**Attribute Name**	**Type**		**Description**
	id	int		ID of the peripheral
	isEnabled	Bool		if the peripheral is enabled
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	draw()	void	Void	Draws the peripheral



LEDArray :: Peripheral

This class is created in order to keep the information of 8 LEDs. The class is inherited from the Peripheral class.				
Attributes	**Attribute Name**	**Type**		**Description**
	ledVector	Vector<LED>		The vector of 8 LEDs
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	sendData()	void	ledID	Sends data to the LED with ledID.
	readInput()	void	Void	Reads the input.
	draw()	void	Void	Draws this peripheral.

LED

This class represents a single LED on the CENG336 Board.				
Attributes	**Attribute Name**	**Type**		**Description**
	ledId	int		The ID of this LED
	ledData	Byte		The data of this LED
	isEnabled	Bool		if this LED is enabled
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	writeData()	void	Byte	Writes the data to this LED.
	readData()	byte	Void	Reads the input.
	draw()	void	Void	Draws this LED.

SevenSegmentDisplayArray :: Peripheral

This class is created in order to keep the information of three "7-Segment Display"s. The class is inherited from the Peripheral class.				
Attributes	**Attribute Name**	**Type**		**Description**
	sevenSegment-DisplayVector	Vector<SevenSegmentDisplay>		The vector of 3 seven segment displays.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	sendData()	void	ssdID	Sends data to the SSD with ssdID.
	readInput()	void	Void	Reads the input.
	draw()	void	Void	Draws this peripheral.

SevenSegmentDisplay

This class represents a single 7-segment display of the CENG336 Board.				
Attributes	**Attribute Name**	**Type**		**Description**
	ssdId	int		The ID of this SSD.
	ssdData	Byte		The data of this SSD.
	isEnabled	Bool		if this SSD is enabled
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	writeData()	void	Byte	Writes the data to this SSD.
	readData()	byte	Void	Reads the input.
	draw()	void	Void	Draws this SSD.

KeyPad :: Peripheral

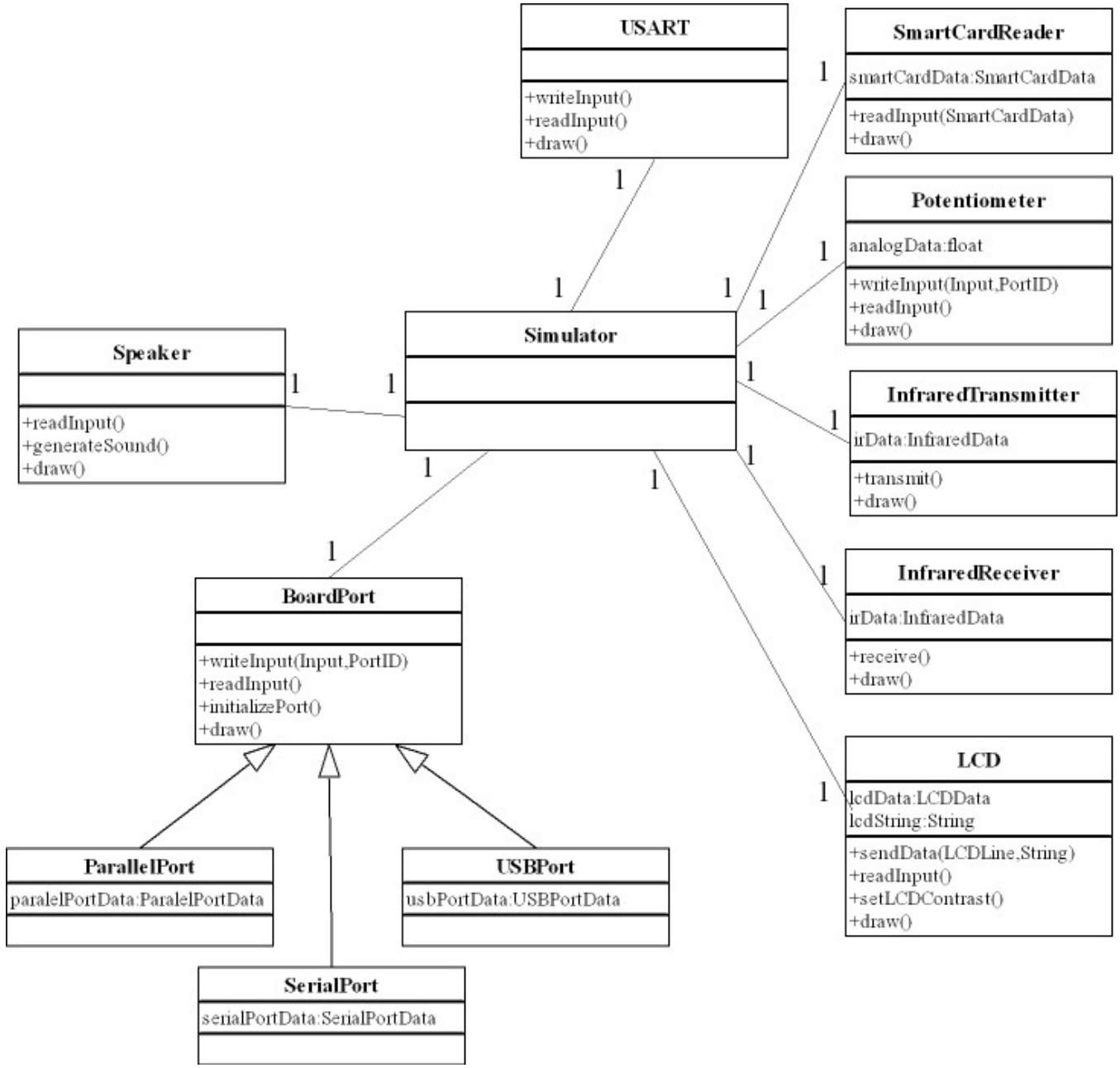
This class represents the Keypad display of the CENG336 Board. The class is inherited from the Peripheral class.				
Attributes	**Attribute Name**	**Type**		**Description**
	pushButtonVector	Vector<PushButton>		The vector of 16 push buttons.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	sendData()	void	buttonID	Sends data to the push button with buttonID.
	readInput()	void	void	Reads the input.
	writeInput()	void	Data, portID	Sends the input *data* to the port with *portID.*
	draw()	void	void	Draws this peripheral.

PushButton

This class represents a single "push button" display of the CENG336 Board.				
Attributes	**Attribute Name**	**Type**		**Description**
	buttonId	buttonID		The ID of this push button.
	buttonData	Int		The data of this button.
	isEnabled	Bool		if this button is enabled
	State	Int		The state of this button
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	readData()	Int	Void	Reads the input.
	draw()	Void	Void	Draws this push button.

ResetButton :: PushButton

This class represents the "reset button" display of the CENG336 Board.				
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	sendResetSignalToPIC()	Void	Void	Sends RESET signal to the PIC.



LCD :: Peripheral

This class represents the LCD display of the CENG336 Board. The class is inherited from the Peripheral class.

Attributes	**Attribute Name**	**Type**		**Description**
	Data	LCDData		The data of the LCD.
	lcdString	String		The string on the LCD.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	sendData()	void	String	Sends data to the LCD.
	readInput()	void	Void	Reads the input.
	setContrast()	void	Float	Changes the contrast of the LCD to the given value.
	draw()	void	Void	Draws this peripheral.

ParallelPort :: BoardPort

This class represents the Parallel Port of the CENG336 Board. The class is inherited from the BoardPort class.				
Attributes	**Attribute Name**	**Type**		**Description**
	Data	ParallelPortData		The data of the port.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	draw()	Void	Void	Draws this peripheral.

SerialPort :: BoardPort

This class represents the Serial Port of the CENG336 Board. The class is inherited from the BoardPort class.				
Attributes	**Attribute Name**	**Type**		**Description**
	Data	SerialPortData		The data of the port.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	draw()	Void	Void	Draws this peripheral.

USBPort :: BoardPort

This class represents the USB Port of the CENG336 Board. The class is inherited from the BoardPort class.				
Attributes	**Attribute Name**	**Type**		**Description**
	Data	USBPortData		The data of the port.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	draw()	Void	Void	Draws this peripheral.

USART :: Peripheral

This class represents "USART(Universal Synchronous Asynchronous Receiver Transmitter)" of the CENG336 Board. The class is inherited from the Peripheral class.				
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	writeInput()	void	Void	Writes data.
	readInput()	void	Void	Reads the input.
	draw()	void	Void	Draws this peripheral.

Speaker :: Peripheral

This class represents the "Speaker" of the CENG336 Board. The class is inherited from the Peripheral class.				
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	readInput()	void	Void	Reads the input.
	generateSound()	void	Data	Generates sound according to the given input.
	draw()	void	Void	Draws this peripheral.

Potentiometer :: Peripheral

This class represents the "Potentiometer" of the CENG336 Board. The class is inherited from the Peripheral class.				
Attributes	**Attribute Name**	**Type**		**Description**
	analogData	float		The analog voltage value of the potentiometer.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	writeInput()	void	Data, PortID	Writes data to the Port with PortID.
	readInput()	float	void	Reads the input.
	draw()	void	void	Draws this peripheral.

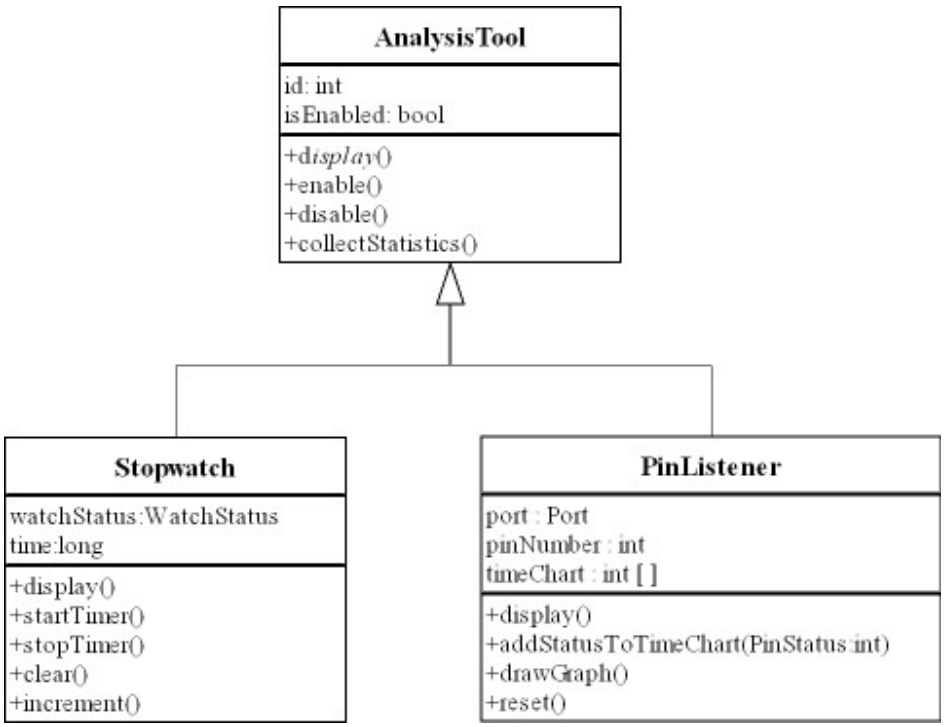
InfraredTransmitter :: Peripheral

This class represents the "Infrared Transmitter" of the CENG336 Board. The class is inherited from the Peripheral class.				
Attributes	**Attribute Name**	**Type**		**Description**
	data	InfraredData		The data of the infrared transmitter.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	transmit()	Void	void	Transmits the data.
	draw()	Void	void	Draws this peripheral.

InfraredReceiver :: Peripheral

This class represents the "Infrared Receiver" of the CENG336 Board. The class is inherited from the Peripheral class.				
Attributes	**Attribute Name**	**Type**		**Description**
	data	InfraredData		The data of the infrared receiver.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	receive()	void	void	Receives the data.
	draw()	void	void	Draws this peripheral.

3.2.4.4. Classes of *analysis* Package



AnalysisTool

This is a general class representing the analysis tools we defined. Specific analysis tools are inherited from this class.				
Attributes	**Attribute Name**	**Type**		**Description**
	id	Int		The ID of the analysis tool.
	isEnabled	bool		If the analysis tool is enabled.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	enable()	void	void	Enables the analysis tool.
	disable()	void	void	Disables the analysis tool.
	display()	void	void	Displays the analysis tool.
	collectStatistics()	void	void	Collects the anaysis results.

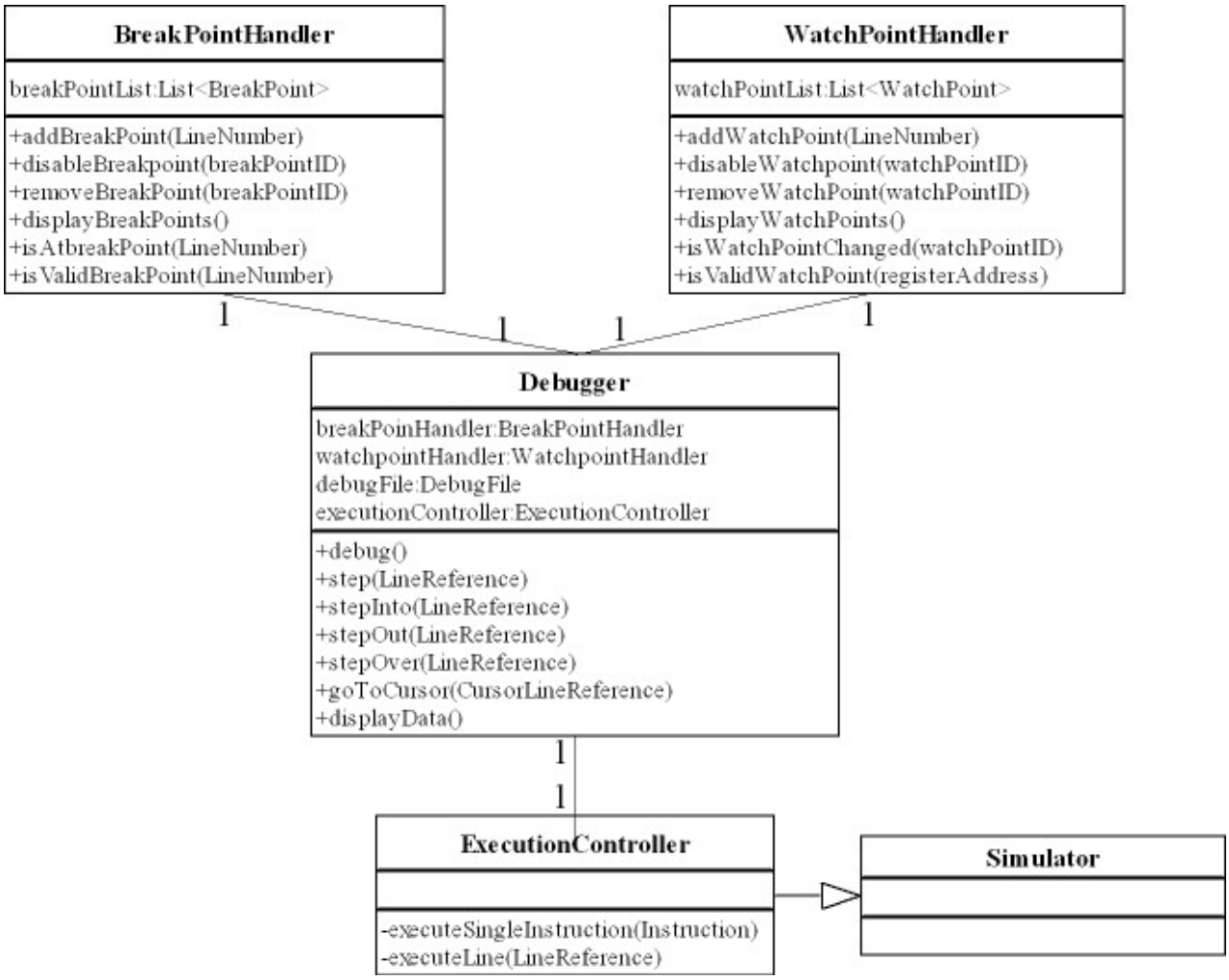
StopWatch :: AnalysisTool

This class represents the "stop watch" property of the analysis tools. The class is inherited from the AnalysisTool class.				
Attributes	**Attribute Name**	**Type**		**Description**
	status	Int		The stop watch status
	time	Long		The time passed during execution
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	startTimer()	void	Void	Starts the timer.
	increment()	void	Void	Increments the timer value.
	stopTimer()	void	Void	Stops the timer.
	clear()	void	void	Resets the timer.
	display()	void	void	Displays the analysis tool.

PinListener :: AnalysisTool

This class represents the "pin listener" property of the analysis tools. The class is inherited from the AnalysisTool class.				
Attributes	**Attribute Name**	**Type**		**Description**
	Port	Port		The Port that the pin belongs to.
	pinNumber	int		The pin number on the Port.
	Status	int		Current status of the pin.
	timeChart	int[]		The array to display the pin value (0 or 5) with respect to time. (Starts from time = 0)
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	addStatusToTimeChart()	void	int	Adds the given status of the pin (0 or 5) to the timeChart.
	drawGraph()	void	void	Draws the timeChart graph.
	reset()	void	void	Resets the pin listener.
	display()	void	void	Displays the analysis tool.

3.2.5. Classes of *debugger* Package



Debugger

This class involves the general attributes and related methods of the debugger.				
Attributes	**Attribute Name**	**Type**		**Description**
	breakPointHandler	BreakPointHandler		The breakpoint handler.
	watchPointHandler	WatchPointHandler		The watchpoint handler.
	debugFile	DebugFile		The file used during debugging process.
	executionController	ExecutionController		The simulator used during debugging process.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	debug()	void	void	Starts the debugging process.
	step()	void	LineReference	Executes one step.
	stepInto()	void	LineReference	Steps into the next block.
	stepOut()	void	LineReference	Steps out of the current block.
	stepOver()	void	LineReference	Steps over the next block.
	gotoCursor()	void	cursorPosition	Executes upto the cursor position.
	displayData()	void	void	Displays the debug data.

ExecutionController :: Simulator

This class involves the methods to control the execution of a program. The class is inherited from the Simulator class.				
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	executeSingleInstruction()	void	Instruction	Executes the the given single instruction.
	executeLine()	void	LineReference	Executes one line in the AsmPlusFile.

BreakPointHandler

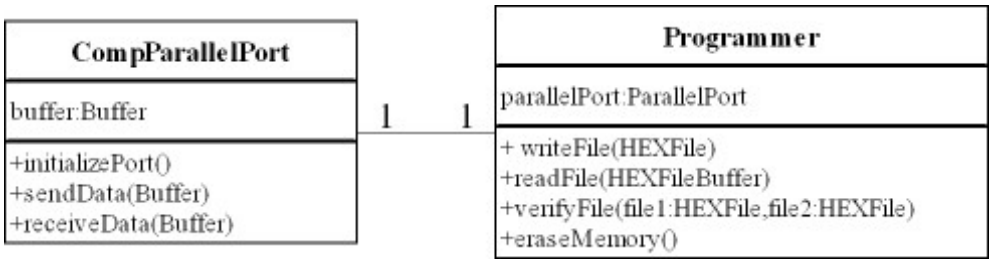
This class involves the necessary attributes and methods related with the breakpoints specified in the debug procedure.				
Attributes	**Attribute Name**	**Type**		**Description**
	BreakPointList	List<BreakPoint>		The list of the breakpoints.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	addBreakPoint()	Void	LineNumber	Adds a break point to the given line.
	disableBreakPoint()	Void	breakPointID	Disables the break point with *breakPointID.*
	removeBreakPoint()	Void	breakPointID	Removes the break point with *breakPointID.*
	displayBreakPoints()	Void	Void	Displays the breakpoints on the editor.
	isAtBreakPoint()	Bool	LineNumber	Returns true if there exists a breakpoint on the line with *lineNumber.*
	isValidBreakPoint()	Bool	LineNumber	Returns true if there exists a valid breakpoint on the line with *lineNumber.*

WatchPointHandler

This class involves the necessary attributes and methods related with the breakpoints specified in the debug procedure.

Attributes	**Attribute Name**	**Type**		**Description**
	watchPointList	List <WatchPoint>		The list of the watchpoints.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	addWatchPoint()	void	Variable	Adds a watch point to the given *variable*.
	disableBreakPoint()	void	watchPointID	Disables the watch point with *watchPointID.*
	removeWatchPoint()	void	watchPointID	Removes the watch point with *watchPointID.*
	displayWatchPoints()	void	Void	Displays the watch points.
	isValidWatchPoint()	bool	registerAddress	Returns true if there exists a watch point associated with the given *registerAddress.*
	isWatchPointChanged()	bool	watchPointID	Returns true if the variable associated with *watchPointID* is changed.

3.2.6. Classes of *programmer* Package



Programmer

This class involves the necessary attributes and methods to program the microcontroller.				
Attributes	**Attribute Name**	**Type**		**Description**
	port	CompParallelPort		The parallel port of the computer to be used for reading/writing programs to the PIC.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	write()	void	HexFile	Writes the hex file to the PIC.
	read()	void	HexFileBuffer	Reads the program on the PIC.
	verify()	void	HexFile, HexFileBuffer	Compares the program on the PIC with the one on the buffer and verifies.
	erase()	void	Void	Erases the program on the PIC.

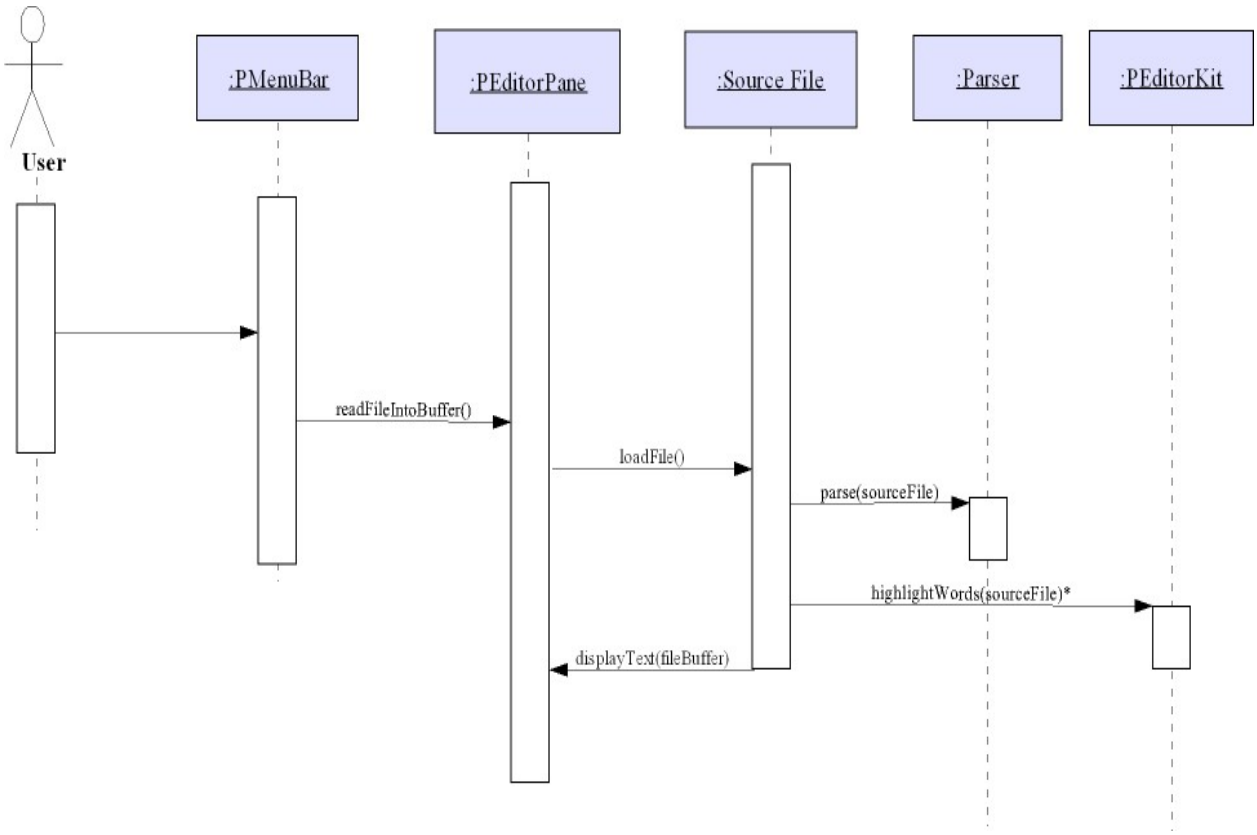
CompParallelPort

This class drives the parallel port of the computer which is going to program the board.				
Attributes	**Attribute Name**	**Type**		**Description**
	portBuffer	Buffer		The buffer to be used for the parallel port of the computer.
Methods	**Method Name**	**Return**	**Arguments**	**Description**
	initialize()	void	void	Initializes the port.
	sendData()	void	void	Sends the data in the buffer to the port.
	receiveData()	void	void	Receives the data from the port into the buffer.

3.3. Flow-Oriented Modeling

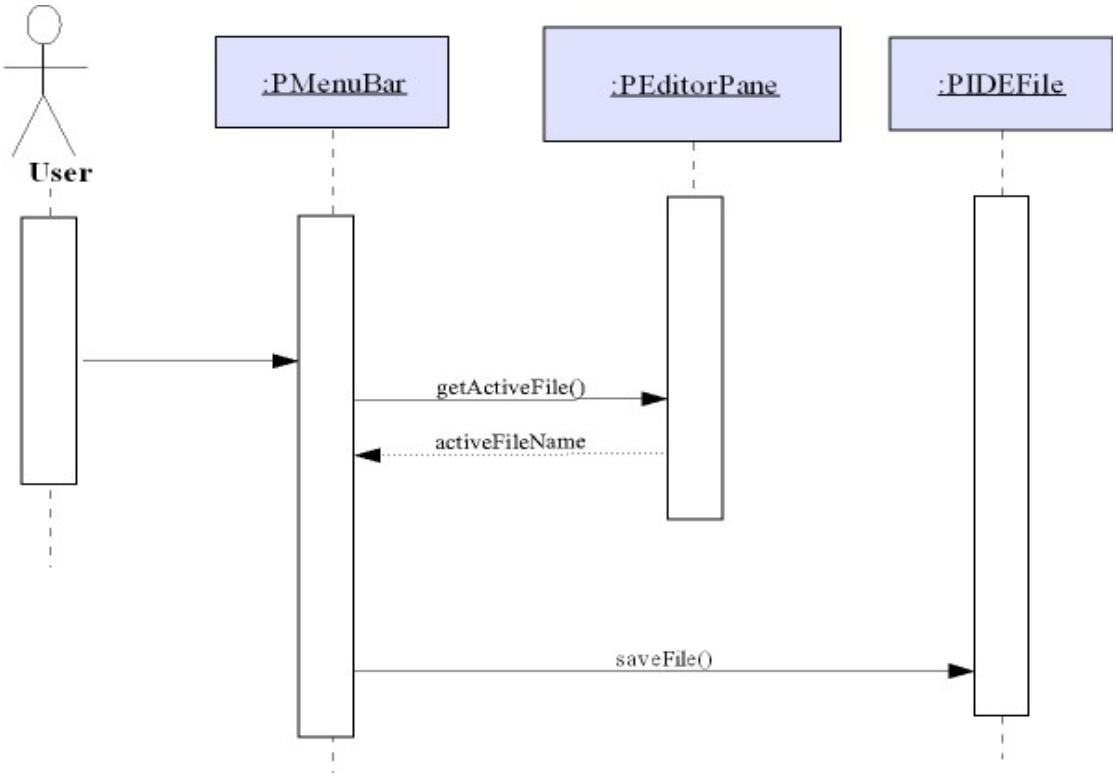
3.3.1. Editor Module

Open File



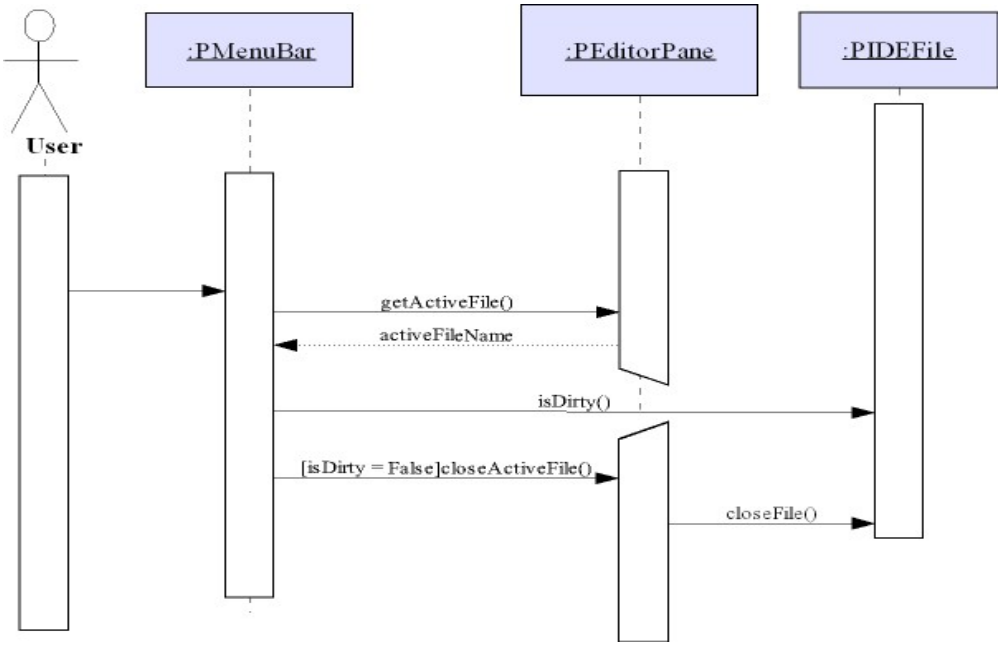
When the user selects "Open File" from the menu, a file selection dialog is shown. As soon as user picks a valid file, it invokes the readFileIntoBuffer() method of the PEditorPane object which reads the text data of the file into its buffer array. For this purpose, this function invokes the loadFile() method of the related source file object. After the actual source file is loaded into memory, it is parsed and tokenized and these tokens are sent to PEditorKit to be colored. After highlighting the source code, it is displayed on the screen.

Save File



When the user selects "Save File" from the menu, the system determines the active file in the current editor panel with the getActiveFile() method of PEditorPane object. Then saveFile() method of the related File object is invoked so that content of the fileBuffer is written into specified file.

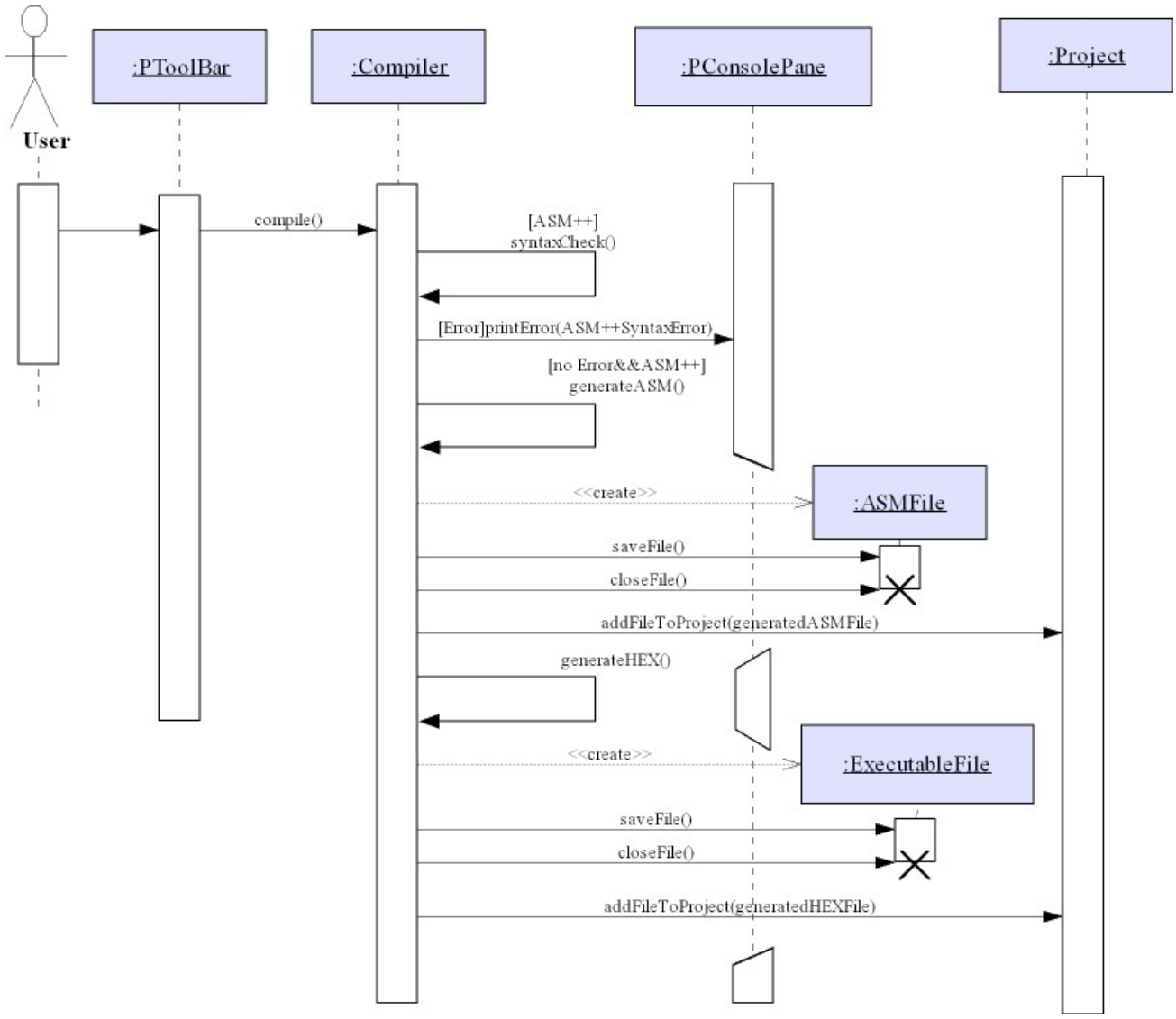
Close File



When the user selects "Close File" from the menu, the system determines the active file with the getActiveFile() method of PEditorPane object and check if the

file is dirty, i.e. any change has been made since the last save operation. If the file is not dirty, the related editor tab of the active file closed. If the file was dirty, then system would warn the user and ask if he would like to save the changes before closing the file.

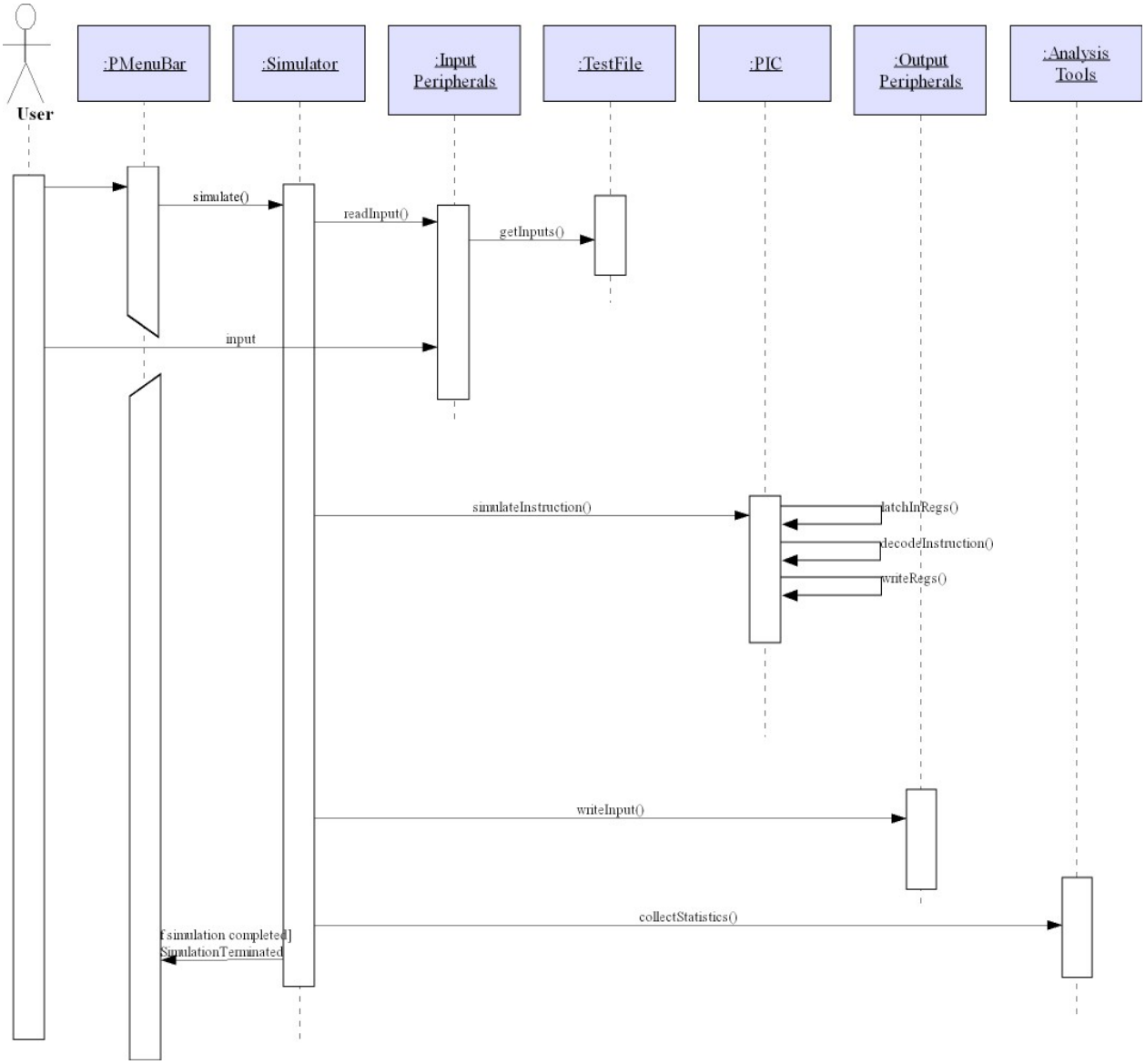
3.3.2. Compile Module



When the user presses "Compile" button on the toolbar, the system invokes the compile() method of the Compiler object. If the file is an ASM++ file, it calls the syntaxCheck() method, which checks if there is an error or not syntactically or not. If there are any errors it prints an error message on the PConsolePane. If there are no errors, it creates an ASMFile object for the intermediate ASM file. The generated ASM file is added to the project workspace.

After generating an ASM file, Compiler object generates the executable file from the ASM file. This file is added into project workspace, too.

3.3.3. Simulate Module



When user selects "Simulate" from the menu, the system invokes the simulate() method of the Simulator module. simulate() method runs until simulation is stopped. In the sequence diagram, operations after simulate() method simulates one clock cycle of the board. In other words, simulation runs in discrete time intervals. During simulation, simulate() keeps simulating the clock cycles.

Simulator module can be run in two different modes. First of them is the direct interaction with the user and the other is using a test file. In either case Simulator sends readInput message to all enabled input peripherals to see if there is an input from a source. Peripherals update their data accordingly and

return. Simulator then sends simulateInstruction() message to PIC which first calls latchInRegs function to take a snapshot of the current registers before simulating a cycle of the PIC. After saving registers, PIC simulates its modules and saves last data with writeRegs() function. After simulating the PIC, Simulator now passes PIC's last state to output peripherals and simulates them. At the end of the cycle, analysis tools such as pin listeners update data with collectStatistics() method. Simulation is stopped by user's request.

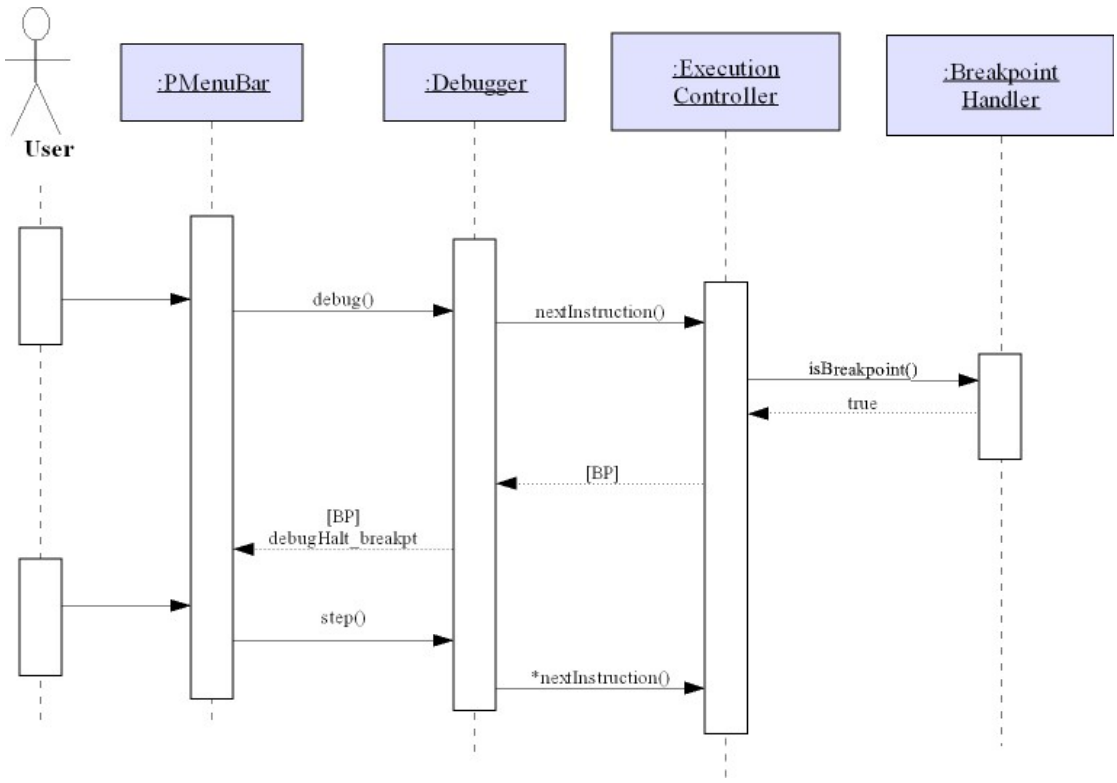
3.3.4. Debugger Module

When the user presses the "Debug" button on the toolbar, the system calls the debug() function of the Debugger object. The Debugger object calls the nextInstruction() method immediately of the ExecutionController object. ExecutionController object calls two functions:

1. isBreakpoint() method of the BreakpointHandler object to see if the current line has any breakpoints. It returns "true" or "false".
2. isWatchpoint() method of the WatchPointHandler object to see if data elements in the current line have any watchpoint. It returns "true" or "false".

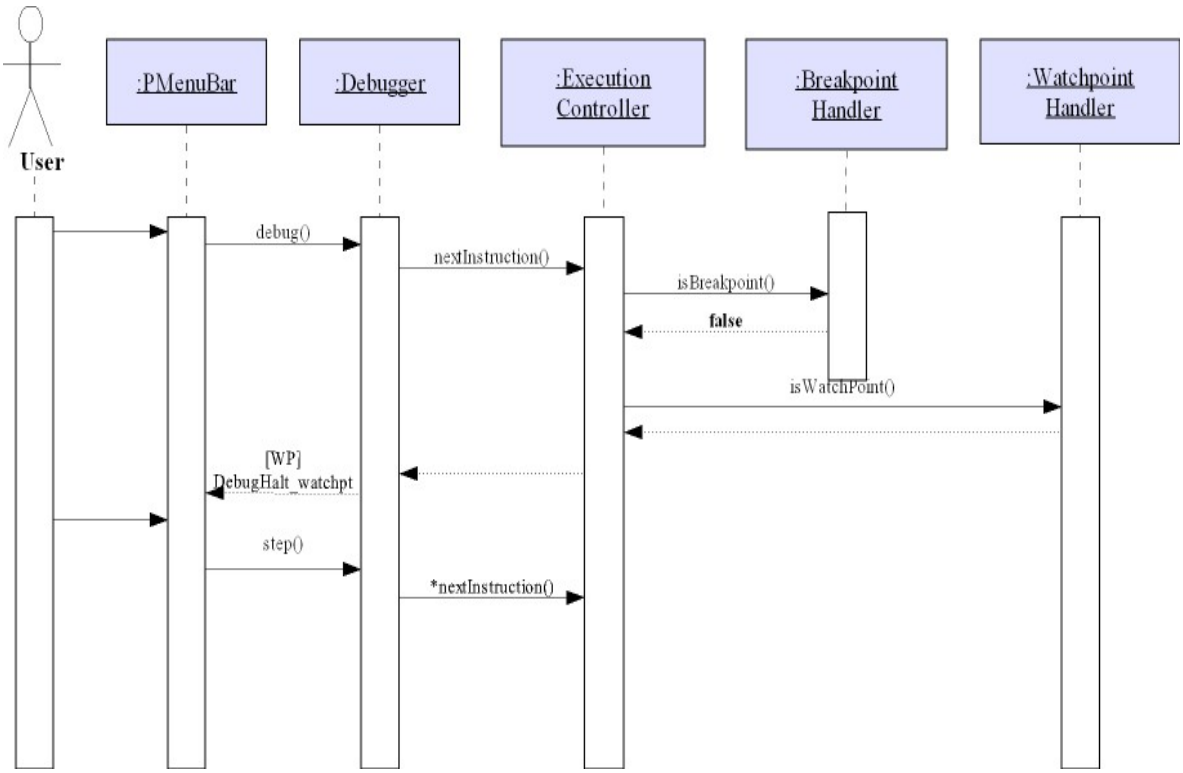
It takes three branches into execution according to the return values of these functions.

Debug at Breakpoint



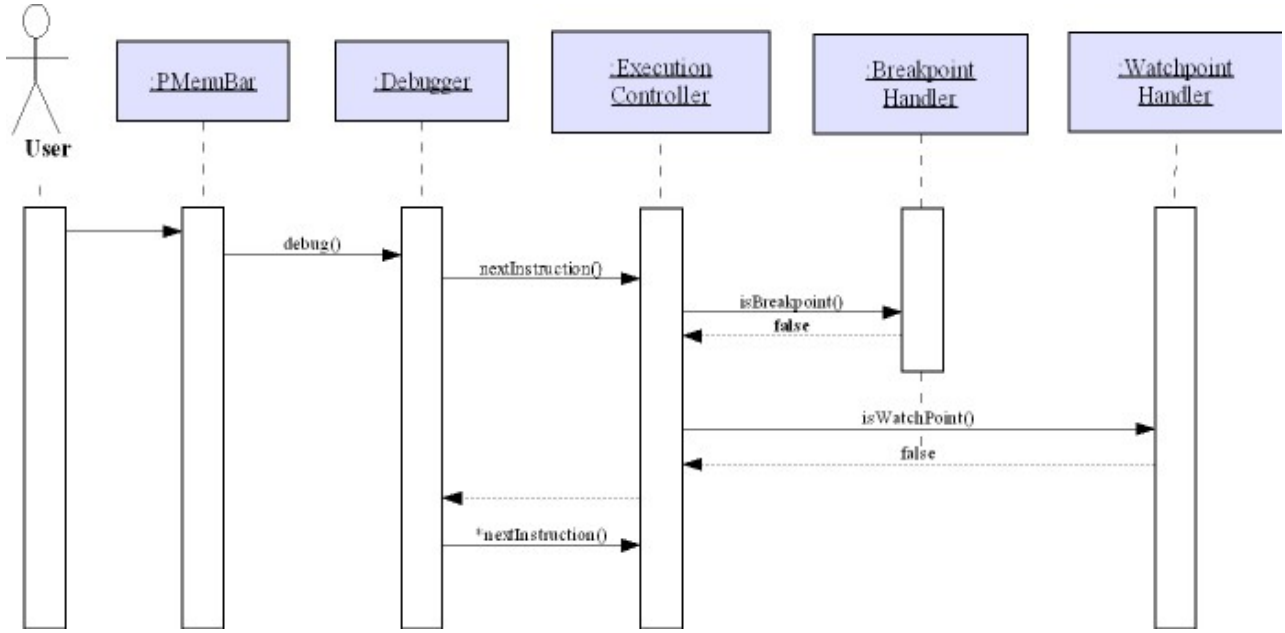
If isBreakpoint() returns true, Execution Control returns the breakpoint information to the Debugger and it informs the system that a breakpoint is reached. So debugger stops simulating and waits for the user interaction. When user steps into the next instruction, the stepCallBack() method is invoked and the Debugger continues with the next instruction.

Debug at Watchpoint



If isBreakpoint() returns false and isWatchpoint() returns true, the Execution Controller returns the watchpoint information to the Debugger and it informs the system that some data that has watchpoint on it has been changed. So debugger stops simulating and waits for the user interaction. When user steps into the next instruction, the stepCallBack() method is invoked and the Debugger continues with the next instruction.

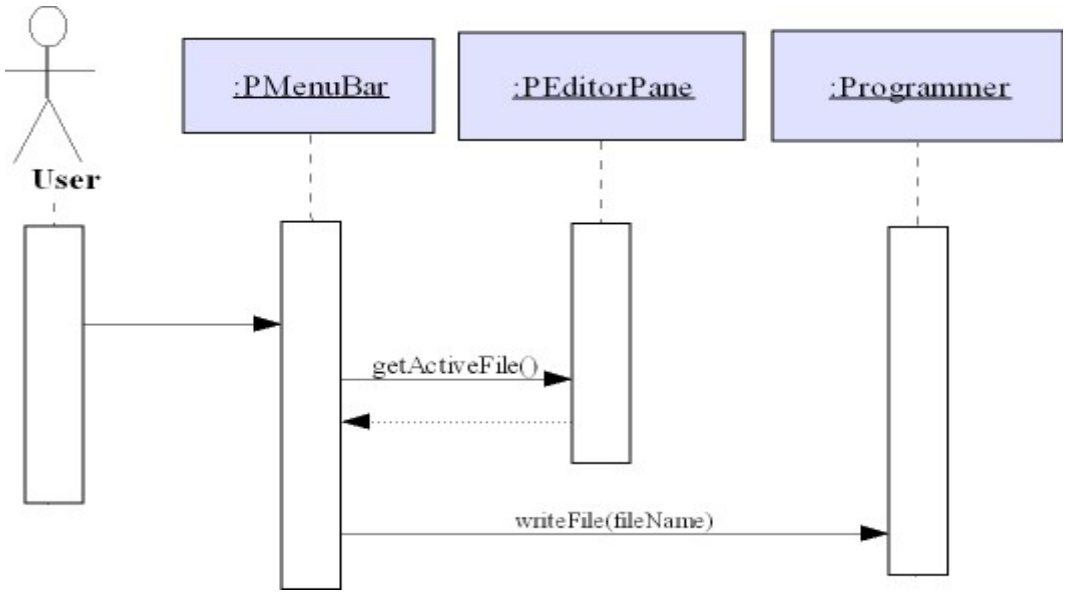
Debug without Breakpoint and Watchpoint



If isBreakpoint() and isWatchpoint() return false, the Debugger steps into the next instruction without any prompt.

3.3.5. PIC Programmer Module

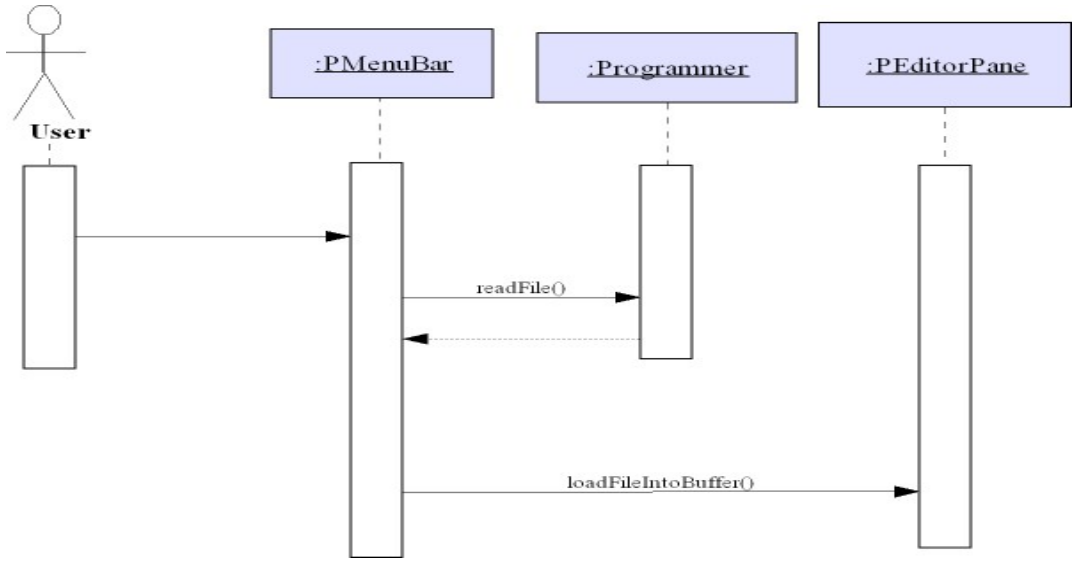
Write



When the user selects "Write" from the menu, the system controls if the active file in the editor is an executable file which is directly passed on to Programmer module. If not, a file browser window pops up so that user can select an

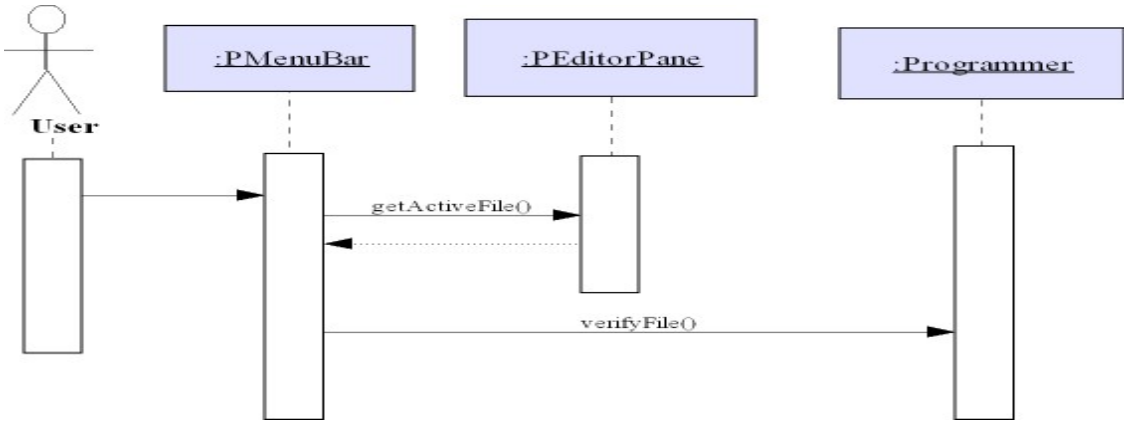
executable file to download to the PIC. Then the module establishes a connection with the board and programs the PIC using parallel port.

Read



When the user selects "Read" from the menu, system invokes readFile method of the Programmer module. Programmer establishes a connection with the board and flash program memory is read by the Programmer. The retrieved file is read to the editor buffer by calling the loadFileIntoBuffer() method of the PEditorPane object.

Verify



When the user selects "Verify" from the menu, the system controls if the active file in the editor is an executable file. If not, a file browser window pops up so that user can select an executable file to compare with the one currently on the PIC. Then the module establishes a connection with the board and compares two files.

4. Graphical User Interface Design

Below in Figure 4.1, the GUI of the PIDE program, showing the menus, toolbars, tabs, workspace view and the status bar can be found.

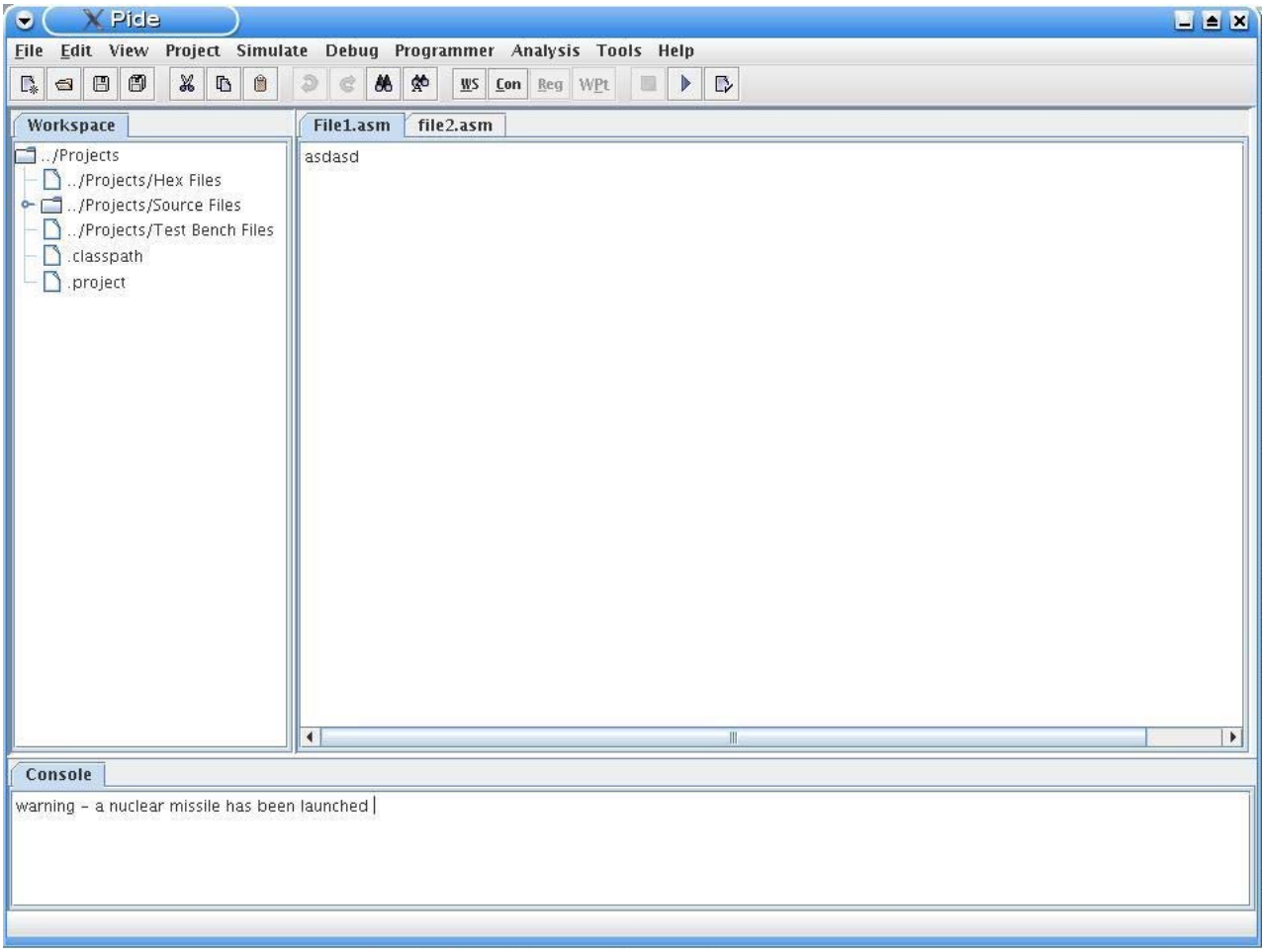


Figure 4.1

Figure 4.1 shows the case with an opened project, and two opened asm files. The workspace view is also present on the left hand side. The program is able to handle multiple opened files using a tabbed view.

In Figure 4.2, the menu bar of the PIDE is shown. The menu items will be explained in detail in the following sections.

Figure 4.2

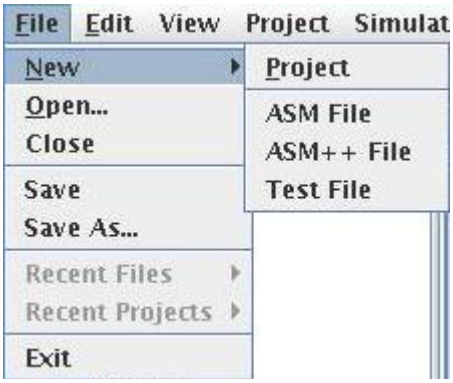
In Figure 4.3, the toolbar of the PIDE is shown. Here exist shortcuts of the frequently used operations in the menu bar.

Figure 4.3

MENUS

There are *File*, *Edit*, *View*, *Project*, *Simulate*, *Debug*, *Programmer*, *Analysis*, *Tools* and *Help* menus in the PIDE program. The operation of each menu item is described below.

FILE MENU



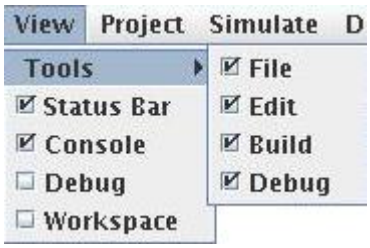
New	**Project**	Create a new project.
	ASM File	Create a new ASM file.
	ASM++ File	Create a new ASM PlusPlus file.
	Test File	Create a new Test file for simulation.
Open…		Open an existing file.
Close		Close the current file.
Save		Save the current file.
Save As…		Save the current file with a different name or save to a different place.
Recent Files		Shows the most recently used files,
Recent Projects		Shows the most recently used projects,
Exit		Quit from the program.

EDIT MENU

Undo	Undo the last action.
Redo	Redo the last undo action.
Cut	Cut the selected item.
Copy	Copy the selected item.
Paste	Paste the last cut or copied item.
Find	Find a given word in the current file.
Replace	Replace the given word with another word.

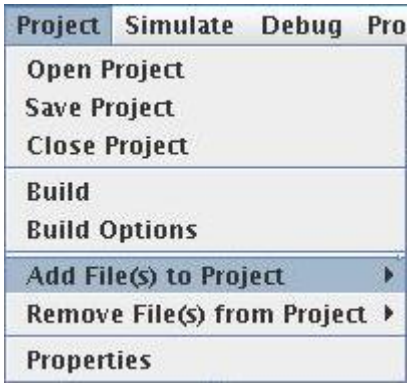
VIEW MENU

Tools	Show/Hide the toolbars of File, Edit, Build and Debug menus.
Status Bar	Show/Hide the status bar.
Console	Show/Hide the console view.
Debug	Show/Hide the debug windows.
Workspace	Show/Hide the workspace view.



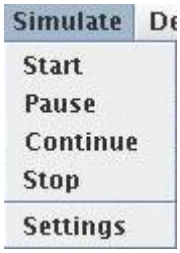
PROJECT MENU

Open Project	Open an existing project.
Save Project	Save the current project.
Close Project	Close the current project.
Build	Build the current project.
Build Options	Change the build options.
Add File to Project	Add a new file to the current project.
Remove File from Project	Remove a file from the current project.
Properties	Change the project properties.



SIMULATE MENU

Start	Start the simulation.
Pause	Pause the simulation.
Continue	Continue the simulation.
Stop	Stop the simulation.
Settings	Change the simulation settings.



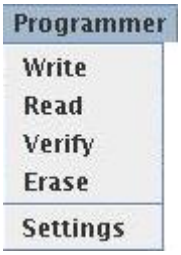
DEBUG MENU

Breakpoint	Add or Remove breakpoints.
Watchpoint	Add or Remove watchpoints.
Start	Start the debugging process.
Step	Execute one step.
Step Into	Step into the next block.
Step Out	Step out of the current block.

Step Over	Step over the next block.
Stop	Stop the debugging process.
Settings	Change the debug settings.

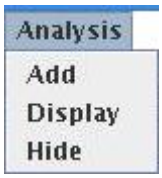
PROGRAMMER MENU

Write	Write the current program onto the PIC.
Read	Read the program in the PIC.
Verify	Verify if the program is written correctly onto the PIC.
Erase	Erase the program written in the PIC.
Settings	Change the programmer settings.



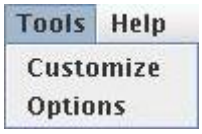
ANALYSIS MENU

Add	Add a new analysis tool.
Display	Display the analysis results.
Hide	Hide the analysis results.



TOOLS MENU

Customize	Customize the program settings.
Options	Change the program options.



HELP MENU

About PIDE…	Show brief information about the program.
Contents	Show the help contents.
Index	Show the help index.
Search	Search a help topic in the help contents.

The settings for different components and operations such as Editor, Debugger, Simulator, Compile or Programmer are grouped in a separate frame which is the "Settings" frame. Below the Programmer settings is shown in Figure 4.4 as an example.

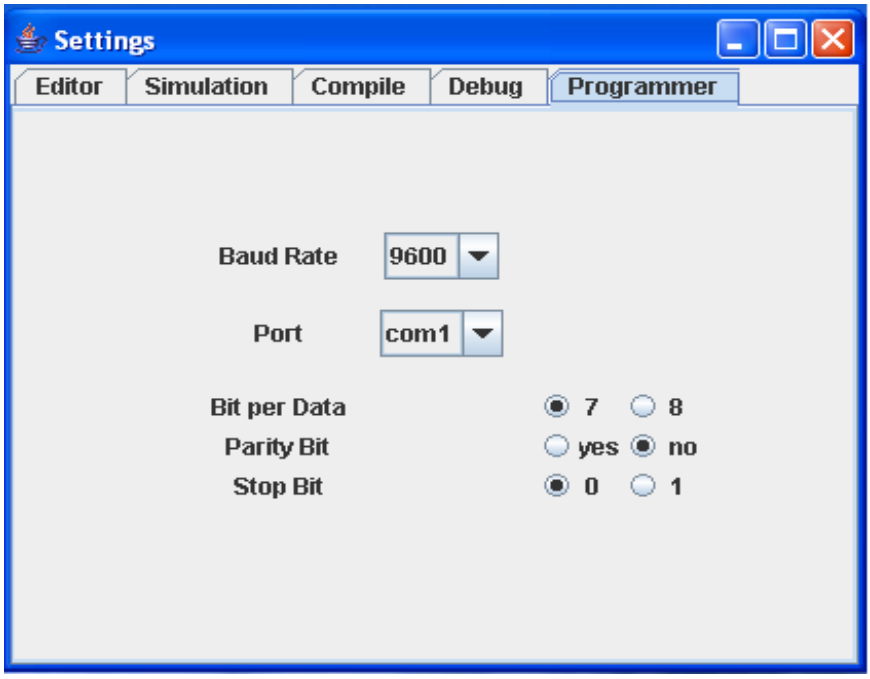


Figure 4.4

5. Components to be Simulated

In this section, the internal structures and implementation details of the board components are described and design strategy of the simulator is given.

5.1. PIC MCU

The PIC microcontroller instruction set contains 35 basic instructions. All of those basic instructions are single word, i.e. 14 bits. They last finite durations, read from some specific registers and update some other specific registers. Therefore, simulations of all 35 instructions are independent and atomic.

It should be emphasized that the accuracy of the simulation is very important in the design. The responses obtained on the actual hardware and the virtual board must be consistent throughout the operation. In order to simulate the hardware which has actually an edge-triggered behavior, the sequential simulator should employ special mechanisms. This is due to the fact that various resources such as registers are shared by multiple modules. Therefore, these shared resources will be implemented within the simulator with local copies inserted into all those sharing modules. Firstly, the input devices will first modify their local registers and then updating the globals. Afterwards, the PIC will simulate itself, updating the local and then global registers. Finally, the output peripherals will be simulated, latching in the global register contents and displaying outputs.

5.1.1. Memory

The memory system of the MCU is composed of FLASH program memory, the RAM Data Memory and the EEPROM Data Memory.

FLASH Program Memory

The program to be uploaded is stored in the Flash Program memory, which has 8KB storage. Each instruction is 14 bits wide. Since the program counter is 13 bits wide, 2^{13} = 8K-words can be addressed in the Flash program memory.

Paging

The FLASH program memory consists of four pages. The address ranges of those four pages are given below.

Page Number	Start Address	End Address
Page 0	0005h	07FFh
Page 1	0800h	0FFFh
Page 2	1000h	17FFh
Page 3	1800h	1FFFh

As a result of the paging system of the program memory, the operations of the jump instructions require special attention. The CALL/GOTO instructions take 11bit arguments, addressing only 2KB of the memory. Actually, the MSB 2 bits of the address are taken from PCLATH<4:3>. Therefore, when a subroutine in another page is to be called, first the PCLATH<4:3> bits should be set accordingly, and then the low order 11 bits should be given to the CALL instruction.

Registers

Register	**Usage**
EEDATA	Data
EEDATH	Data
EEADR	Address LSBs
EEADRH	Address MSBs (0000h-1FFFh)
EECON1	controls
EECON2	controls

Read and Write Operations

Data read operation from the FLASH memory is performed as single word read and data write operation is performed as four word block write.

Read

1. Write address to EEADRH and EEADR
2. Set EEPGD
3. Set RD
4. Wait for 2 cycles idle (those statements are ignored)
5. Read from EEDATH and EEDAT

Write

A write operation to the FLASH program memory can only be performed if not write-protected mode is selected, as defined in device configuration word bits WRT<1:0>

Data is written in four word blocks, where a block is four words with sequential addresses. These four words are identified by EEADR<1:0> bits.

Load ALL 4 buffer registers with order 00-01-10-11:

1. Write address to EEADRH and EEADR
2. Write data to EEDATH and EEDATA
3. Set EEPGD
4. Write 55h to EECON2
5. Write AAh to EECON2
6. Set WR
7. Wait for 2 cycles
8. When last one is written, data is transferred from buffers to FLASH.
9. Then, processor waits for 4ms for the write to be completed.

RAM Data Memory

The RAM data memory is 512B, containing the special purpose registers and general purpose registers (368B).

Bank System

The RAM Data memory is comprised of 4 banks. Bank selection is performed by means of RP1 (Status<5>) and RP2 (Status<6>) bits. Data memory registers can be divided into two groups. First group is the special purpose registers. They are used to control the inputs, outputs and other PIC functionalities. The other group is the general purpose registers. They are simply used as data storage. Data memory size is 128 Bytes/Bank (128 = 0x7F).

An important note should be added here. Since the central processing unit of the PIC microcontroller has a very limited RISC architecture core, it has no special registers in it. Also the memory read/write speed is the same as the registers inside the CPU. As a result, the memory of the PIC is used just as registers. Therefore, Microchip refers the memory words as registers and in this report from this point forward, the data memory words will be referred as registers.

The distribution of the data memory space is given in the figure. As can be seen from the above distribution of the registers, the first portions of all four banks are reserved for special purpose registers and the rest for general purpose registers.

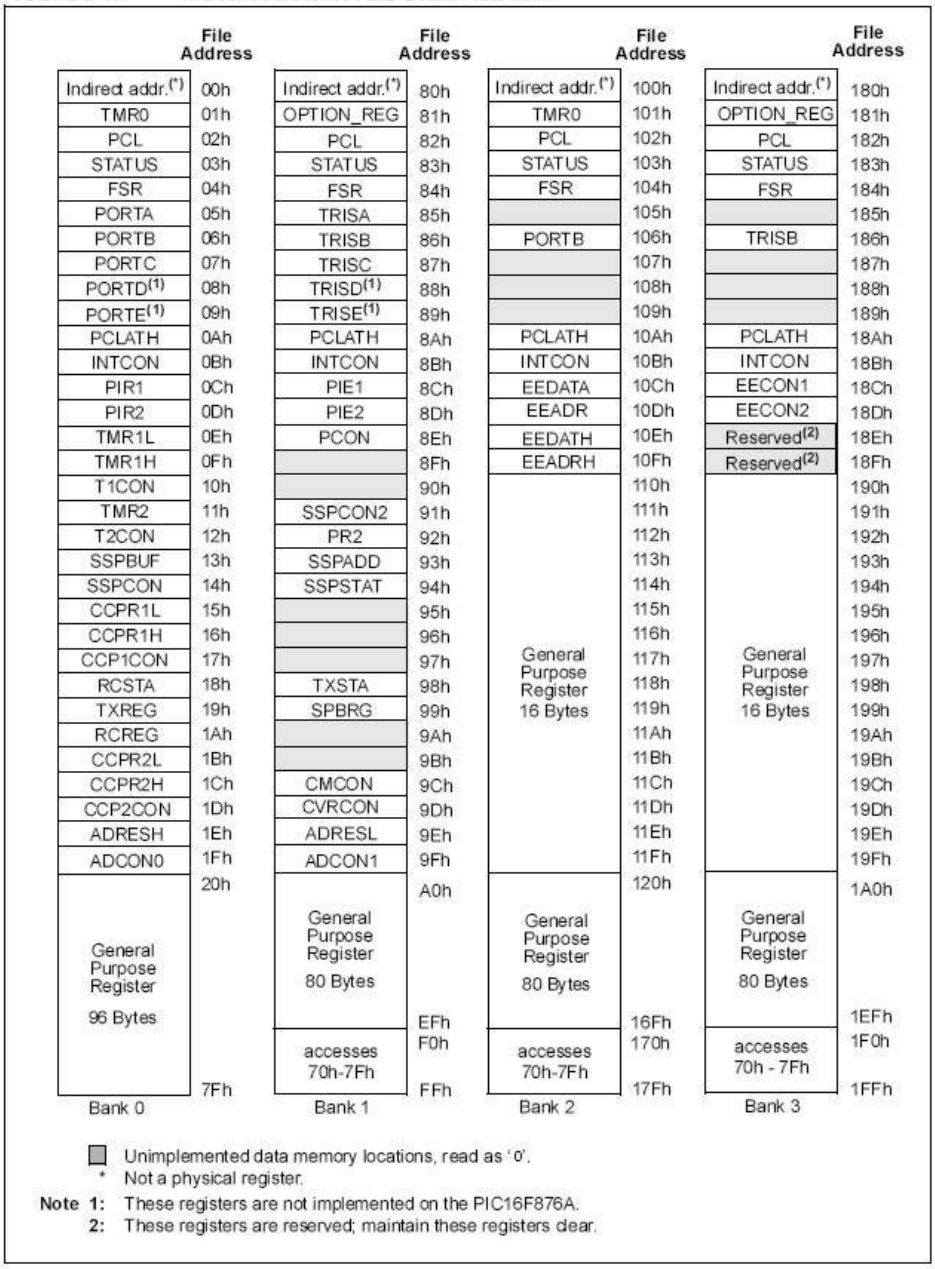
The bitwise explanation of the special purpose registers are given in the PIC 16F877 datasheet by Microchip.

A careful examination of the above data memory address space gives us why Microchip defines the data memory “up to 368 Bytes”. The General Purpose Registers are totally 96+80+16+80+16+80 = 368 Bytes.

Among the special purpose registers, the some registers are of special interest. Those registers are STATUS, OPTION, and INTCON. STATUS register is controlled to switch between the banks, Time-out, Power-down modes and carry/borrow control of arithmetic operations. OPTION register is used to enable PORTB internal weak pull ups, Interrupt enabling, timer source and edge and prescale selections. INTCON register is used to configure the interrupts in the system. Global, peripheral, timer, external, portB interrupts are enabled and the flags are read/cleared from this register. There are also PIE1, PIR1, PIE2, PIR2 registers for enabling peripheral interrupts and their flags.

The PCON register contains the flags for different types of reset operations such as power-on reset, watchdog reset external reset and brown-out reset.

FIGURE 2-3: PIC16F876A/877A REGISTER FILE MAP



Unimplemented data memory locations, read as '0'.
* Not a physical register.

Note 1: These registers are not implemented on the PIC16F876A.
2: These registers are reserved; maintain these registers clear.

Indirect Addressing

Indirect addressing is accomplished by means of INDF virtual register name. When INDF is used as the target address, actually the address pointed by the FSR (File Select Register) register is accessed. 8 bits in FSR register and 1 IRP bit give 9 bits to address the overall 2KByte data memory (000h – 1FFh).

EEPROM Data Memory

The EEPROM data memory has 256Bytes of storage and is the non-volatile data storage system.

Register	Data Memory
EEDATA	8 bit data
EEADR	Address (00h-FFh)
EECON1	Controls
EECON2	Controls
PIR2	flags

Data read operation from the EEPROM memory is performed as single byte read and data write operation is performed as single byte write. The EEPROM data memory is not directly addressed, but is accessed indirectly via special registers.

EECON1 Register Contents

EEPGD=0	Data
EEPGD=1	Program
RD	read , can only be set by user; reset by hardware
WR	write , can only be set by user; reset by hardware
WREN	write enable
WRERR	write error when there's a MCLR or WDT reset

PIR2 Register Contents

EEIF	Write complete interrupt flag

Read Operation

1. Write address to EEADR
2. Clear EEPGD
3. Set RD
4. Next cycle, data is ready at EEDATA, so next instruction can read it

Write

WR inhibited from being set if WREN is cleared

1. Write address to EEADR.
2. Write 8-bit data EEDATA
3. Clear EEPGD
4. Set WREN
5. Disable interrupts (if enabled).
6. Execute the special five instruction sequence:
    i. Write 55h to EECON2 in two steps (first to W, then to EECON2)

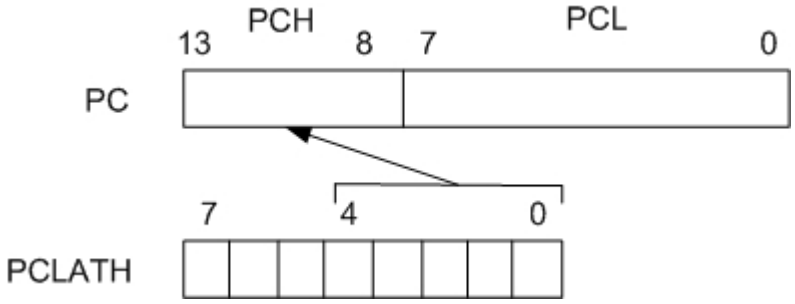
ii. Write AAh to EECON2 in two steps (first to W, then to EECON2)
iii. Set the WR bit

7. Enable interrupts (if using interrupts).
8. Clear the WREN.
9. At the completion of the write cycle, the WR bit is cleared and the EEIF interrupt flag bit is set. (EEIF must be cleared by firmware.)

Program Counter

The program counter (PC) of the Microcontroller is a part of the data memory. The value inside the PC shows the next instruction to be executed in the program memory. The PC 13 bits, and is held in two registers.

- 8 LSBs (<7:0>) are in PCL register, readable and writable.
- MSB 5 bits (<12:8>) are copied from PCLATH register (<4:0>) on a "write to PC" instruction such as "ADDWF PCL".



PC Stack

Related to the PC, the stack is of primary importance. Stack is used to store the current value of the PC in case of a subroutine/function call, to be able to proceed with normal operation upon return. The user cannot access (i.e. read or modify) the stack.

- Stack is 8 PC words (13 bits) deep.
- Stack pointer is not readable / writable
- Stack is circular, i.e. a 9^{th} write overwrites stack address 0.

5.1.2. PORTS

There are five ports on the microcontroller. These ports are used for various purposes, but mostly for digital I/O. The names of the ports and the number of pins on each are PORTA (6), PORTB (8), PORTC (8), PORTD (8) and PORTE (3).

Port Name	Pins	Connected Modules
PORTA	5	Digital Input/Output A/D Converter(default) Comparator SPI Timer0
PORTB	8	Digital Input/Output External Interrupt Weak internal pull-up Interrupt on Change
PORTC	8	Digital Input/Output Timer1 PWM 1-2 SPI I2C USART
PORTD	8	Digital Input/Output Parallel Slave Port
PORTE	3	Digital I/O A/D Converter Parallel Slave Port

Interrupts

There are 15 sources of interrupts in the system. Therefore, that number of interrupt vectors will be used to select the address to be jumped onto in case of an interrupt. Among the most important interrupt vectors, the reset vector of the system resides in the address 0000h and the external interrupt vector in 0004h.

5.1.3. Parallel Slave Port

Parallel Slave Port registers and usage:

Set TRISE<2:0> for inputs

ADCON1<3:0> for digital I/O not analog I/O

Write with WR low and CS low, when any one becomes high, IBF flag is set, and PSPIF interrupt flag is set

Read PORTD to clear IBF

If a second write before read, IBOV is set

Read with RD low and CS low, OBF is cleared, when any one becomes high, PSPIF interrupt flag is set, OBF low until data is written

5.1.4. Analog to Digital Converter

The conversion of an analog input signal results in a corresponding 10-bit digital number. The A/D module has high and low-voltage reference input that is software selectable to some combination of VDD, VSS, RA2 or RA3.

The ADRESH and ADRESL registers contain the 10-bit result of the A/D conversion. When the A/D conversion is complete, the result is loaded into this A/D Result register pair, the GO/DONE bit (ADCON0<2>) is cleared and the A/D interrupt flag bit ADIF is set. The block diagram of the A/D module is shown in Figure 11-1.

Clearing the GO/DONE bit during a conversion will abort the current conversion. The A/D input pins must be configured as input pins via the TRIS register to be used as analog inputs.

INTCON	Interrupt Enable
PIR1	Interrupt flag
PIE1	Interrupt enable
ADRESH	Conversion Result MSBs (or LSBs)
ADRESL	Conversion Result LSBs (or MSBs)
ADCON0	Analog input channel selection Conversion clock selection Conversion flag A/D enable
ADCON1	AD port configuration Result format selection
TRISA	Pin directions
PORTA	Analog input port
TRISE	Pin directions
PORTE	Analog input port

5.1.5. Other Features of the MCU

Timer0, 8Bit timer/counter with 8Bit prescaler
Timer1, 16Bit timer/counter with prescaler
Timer2
Capture-Compare-PWM Modules
SSP, Synchronous Serial Port
SPI, serial Peripheral Interface
I2C
USART, Universal Synchronous / Asynchronous Receiver Transmitter (9-bit)
BOR, Brown Out Reset
Analog Comparator Module
WDT, Watchdog Timer
Sleep Mode

5.2. Peripherals

The CEng 336 board is a complete evaluation board that contains various devices on it. These devices can be classified into two with respect to their usage, input devices and output devices. The list of the peripherals on the board are given below with their brief explanations.

5.2.1. Input Peripherals

Parallel Port

Parallel port (LPT) is the port that is used for programming the microcontroller on the evaluation board. This port can be used for parallel communication, such as PSP mode, or for serial communication, either synchronous or asynchronous.

Serial Port

Serial port connection, i.e. RS232, is used for asynchronous serial data transfer between other devices and the microcontroller.

USB Port

The USB port is a high speed serial communications interface. For PIC applications, in fact the speed of the USB port is very high, however since in the

recent PCs, the serial communications port is being replaced with the USB ports, the controller should be able to communicate using this protocol.

Smart Card Reader

Smart card reader provides extra storage capability to the system. Since the storage capacity of the EEPROM on the MCU is limited, some extra storage may be necessary. The addressing and read/write operation of the reader should be modeled in the system.

Infrared Transmitter and Receiver

Infrared communication is included on the board to be used for special purpose applications. The system is internally analog and requires special modelling.

Keypad

There are 16 pushbuttons on the evaluation board. The pushbuttons are active high buttons, pulled low during normal operation.

Reset Pushbutton

The reset pushbutton, being active low, is directly connected to the reset of the microcontroller. An MCLR signal is asserted with this input.

5.2.2. Output Peripherals

Led Array

A light emmiting diode (LED) is nothing but a semiconductor device that emits light when given logic high value.

Seven Segment Display Array

A collection of LEDs, arranged in a format that will enable the display of alphanumeric characters is called a seven segment display. On the CEng336 board, there are three of those devices, forming an array.

LCD

Using light emmiting diodes for displaying data is clearly not the best method. Seven segment displays improve the user interface a little but still, it is very old fashioned. Newest systems always include some LCD components as the interface. These devices latch in the data entered, decode the characters and display them on their screen. Moving the cursor on the LCD and deleting are some special operations available on most of the off-the-shelf LCD modules.

Speaker

A speaker is a source of accoustic waves. The input signal is analog and the frequency/intensity of the accoustiv waves is determined by the input waveform characteristics.

6. Language Specifications

6.1. ASM++ Language Format

A Simple Language: ASM++

We have decided to define a new language, which is simply an improvement on assembly language, including some new keyword definitions and introducing some high level language concepts such as function calls and variable definitions. The name of the language is ASM++ (ASM plus plus), and the file extension is ".asmpp".

GENERAL SPECIFICATIONS

- ASM++ is not case sensitive. Upper-case letters and lower-case letters are not considered to be distinct in any token, including reserved words.
- White spaces (space character, tab character and end-of-line) serve to separate tokens; otherwise, they are ignored.
- No token can extend past end-of-line.
- Spaces may not appear inside any token except character and string literals.
- A comment begins with two forward slashes (as in C++) or with a semicolon (as in assembly language) and extends to end of line.
- There cannot be more than one statement in a line.
- No semicolons exist at the end of statements. (In fact, that does not matter since, after a semicolon, the rest of the line is considered as comment.)

IDENTIFIERS

Identifiers start with a letter or an underscore and contain letters, underscores and digits. An identifier must fit on a single line and its first 20 characters are significant.

RESERVED WORDS

The following keywords are reserved in ASM++:

addff	subff	addwff	subwff	swapff	
iorwff	andwff	xorwff	movff		
if	else	for	while	do	continue
break	function	return	define	var	array

OTHER TOKENS (DELIMITERS AND OPERATORS)

One-character delimiters:	: ; , ( ) EOF
One-character operators:	! < = > '
Two-character delimiters:	//
Two-character operators:	== != >= <= && ||

MACROS

Macros are introduced by declarations of the form:

define *name number*

VARIABLES

Variables are introduced by declarations of the form:

var *var_name*
or
var *var_name var_address*

The first declaration reserves one of the predefined addresses from the data memory of the PIC for that variable. The exact locations will be provided to the user in the user manual. While using the first declaration method, it is the user's responsibility to ensure that the correct bank is selected, before using that variable. The second declaration reserves the given *address* for that variable.

Examples:

var abc

var def 0x121

ARRAYS

Arrays are introduced by declarations of the form:

array *array_name*(*array_length*)

or

array *array_name*(*array_length, start_address*)

Similar to the variable declaration, arrays can be declared by specifying a starting address or by using the predefined constant starting addresses. Using the second declaration method, an array of length *array_length* will be reserved starting from the address *start_address* from the data memory of the PIC.

For example:

array abc(10, 0x5510)

LITERALS

A literal consists of a sequence of one or more digits in decimal, binary or hexadecimal format.

A character literal is a single character enclosed by a pair of apostrophes (sometimes called "single quotes".) Examples include 'A', 'x', and '''. A character literal is distinct from a string literal of length one.

There is nothing like string literal.

EXPRESSIONS

In ASM++, expressions are defined as below:

<expr> : <expr1> && <expr1> | <expr1> || <expr1>
<expr1> : <label> == <label> | <label> != <label> |
<label> > <label> | <label> < <label> |
<label> >= <label> | <label> <= <label>

<label> : <address> | <const>
<const> : <variable> | <number>

<address> is the memory addresses in the PIC, with the form 0x045,
<variable> is the variable declared using **var** or **define** keywords,
<number> is the number represented in binary, decimal or hexadecimal format.
For binary operations including the assignment operation, both operands must be of the same type for consistency.

SHORT CIRCUITING

Logical operators AND and OR use short-circuit evaluation. This means that, as soon as the truth value can be determined, evaluation stops. For example, if the first operand of an AND evaluates to **false**, the expression will evaluate to **false**, no matter what the second operand is; i.e. the second operand is not even evaluated. Similarly, if the first operand of an OR evaluates to **true**, the second is not evaluated.

STATEMENTS

- **Assignment statement**
  "**=**" is the assignment operator.
  For example:
  var a 0x121
  a = '0x1C4'

- **If statement**
  An if-statement can be used alone or together with an else-statement. The curly braces are compulsory regardless of the number of statements inside the if-block. The syntax of an if-else statement is as follows:

```
define MAX 100
define MIN 0
.............
if (x > MAX)
{
        goto hede
}
else if(x < MIN)
{
        goto hodo
}
```

- **Loop Statements**

  The compiler will support **while**, **do-while** and **for** loops. The curly braces are compulsory regardless of the number of statements inside the loop. **continue** and **break** instructions are also available with the same effects as in C language. The syntaxes of the loop statements are as follows:

```
while (hede)
{
        .................
        .................
}


for (expr1; expr2; expr3)
{
        .................
}


do
{
        .................
}
while (hede)
```

FUNCTION DEFINITIONS

The ASM++ language will provide function calls. Function calls can be limitedly nested. A sample function definition is as follows:

```
function func_name(parameter1, parameter2)
{
        .............
        .............
        return var1
}
```

Predefined memory addresses will be reserved for parameter passing and for function return values. User will be informed about the memory address usage scheme by means of the user manual.

COMMENTS

The comments are specified by a semicolon or two forward slashes. It will comment out the characters until the end of line.

EXTENDED INSTRUCTION SET

PIDE program will provide a bunch of new instructions together with the basic PIC instruction set. Using these new instructions, it will be possible to do arithmetic operations between two file registers without using the working register WREG. These instructions are:

addff v1 v2 : (V1 <— V1 + V2)
Adds the value of v2 to v1, and writes the result back to v1.

subff v1 v2 : (V1 <— V1 - V2)
Subtracts the value of v2 from v1, and writes the result back to v1.

addwff : (W <— V1 + V2)
Adds the value of v2 to v1, and writes the result to WREG.

subwff : (W <— V1 - V2)

Subtracts the value of v2 from v1, and writes the result to WREG.

swapff : (Temp <— V1, V1 <— V2, V2 <— Temp)

Swaps the values of v1 and v2.

iorwff : (V1 <— V1 OR V2)

Takes the OR of v1 and v2, and writes the result to WREG.

andwff : (V1 <— V1 AND V2)

Takes the AND of v1 and v2, and writes the result to WREG.

xorwff : (V1 <— V1 XOR V2)

Takes the XOR of v1 and v2, and writes the result to WREG.

movff : (V2 <— V1)

Copies the value of v1 to v2.

6.2. Test Bench File Language Format

During the simulation of a source file, the user will want to enter various inputs to the system. The input devices on the board are communication ports, keypad, pushbuttons and pots. Using a test bench file, the user can state the exact time instants that the inputs from these devices will be modified, e.g. a reset signal may be asserted for a period. Test bench files will release the burden of entering the inputs to peripherals at correct instants. This is especially useful in the case of high frequency input requirements.

Test bench file can control the system inputs in two different modes. In the Peripheral mode, the user may control the timing of the inputs to the peripheral devices. Alternatively, in the PIC mode, the user may choose to directly access the pins of the microcontroller. The mode selection is performed by <ModeName> tag. A test file may contain only one mode selection tag.

The format of the test bench files is given below. The file should have ".test" extension.

timescale <time unit>

<PIC>
    #<time> **PORT**<Port Name>**.PIN**< Pin No> = <Expression1>
    #<time> **PORT**<Port Name> = <expression2>

    always #<time> **PORT**<Port Name>**.PIN**<Pin No> = <expression1>
    always #<time> **PORT**<Port Name> = <expression2>

    #<time> **$finish**

timescale <time unit>

<PERIPHERAL>
    #<time> <DeviceName>**.PIN**<Pin No> = <expression3>
    #<time> <DeviceName> = <Expression4>

    always #<time> <DeviceName>**.PIN**<Pin No> = <expression3>
    always #<time> <DeviceName> = <Expression4>

    #<time> **$finish**

Indentation is not important, since the parser ignores white spaces. The instructions are not case-sensitive.

The language for the Peripheral and PIC modes are defined below.

For Peripheral Mode:

```
<Expression3> = 0 | 1 | <DeviceName>.PIN<Pin No>
                      | ~<DeviceName>.PIN<Pin No>

<Expression4> = <word>      | <DeviceName>  + <CONST>
                            | <DeviceName>  - <CONST>

<Device Name> = LPT | RS232 | USB | Keypad | Reset
```

PIC Mode:

```
<Expression1> = 0 | 1 | PORT<Port Name>.PIN<Pin No>
                      | ~PORT<Port Name>.PIN<Pin No>

<Expression2> = <byte>      | PORT<Port Name> + <CONST>
                            | PORT<Port Name> - <CONST>
                            | PORT<Port Name>

Port Name = PORTA | PORTB | PORTC | PORTD | PORTE
```

Example Files

For PIC Mode:

```
timescale <1ms>

<PIC>
        #0 PORTA = 0
        #0 PORTB = 0

        always #10 PORTA.2 = ~PORTA.2
        always #100 PORTB = PORTB + 1

#<1000> $finish
```

For Peripheral Mode:

```
<PERIPHERAL>
        #0 Keypad = 0
        #0 Reset = 1
        #5 Reset = 0

        #10 Keypad.PIN5 = 0
        #10 Keypad.PIN2 = ~Keypad.PIN2
        always #100 Keypad.PIN3 = ~Keypad.PIN3

#<1000> $finish
```

7. File Formats

7.1. System File Format

PIDE is designed to be customizable and intelligent in the sense that user is able to work with the last saved configuration. PIDE saves all configuration data into a system file "pide.sys". System file holds data of users' preferences and default settings about the overall program execution such as coloring schemas, font type and size of the editor, record of recent files and projects. Access to system file will be restricted and the file will be hidden.

System file consists of informative comments (comment token is '#') followed by default settings of the system and user defined settings. If somehow some user information is missing, program handles it by loading the default setting. However, if both user setting and default setting of the same preference is missing, user may add a line into the file defining a default value for the preference. Users can refer to the system manual for system file specifications, but user is discouraged to change the system file. Below is an example system file.

```
#PIDE vers. 1.0
#Install date: 18.9.2007 12:33:48
#You are discouraged to change the values in this file since
#it may cause unexpected program behavior. For an emergency case,
#please refer to system manual.
#Default System Settings
...
...
#Default Editor Settings
text_color: 0 0 0
background_color: 255 255 255
text_size: 12
...
```

```
...
#Latest System Settings
#set on 26.9.2007
#Recent files
recent_documents:
>./source/heat_sensor.asmpp
>./myLib/a2dcalculate.ah
>./testcase.test
recent_projects:
>./projects/heat sensor/
#Editor Settings
text_color: 0 0 5
background_color: 255 255 250
text_size: 14
...
```

7.2. Project File Format

PIDE is designed to be able to create projects and save workspaces for a better IDE experience. PIDE saves all necessary information in a file <project_name>.pde to recreate a previously used workspace. "pde" is the PIDE project save file extension. Each project has a pde file under its project folder. Below are the specifications and format of the project file.

Project Description in Project File

Project files include a project description section at the beginning. It includes version of PIDE, name of the project, user/corporate name, creation and last modification dates of the project and description of the project if available. Each description is leaded by a keyword and followed by a new line. Project description can span several lines with project description token (#) at the beginning of each line. Below is an example of the project description section.

```
#PIDE 1.0- PIC Integrated Development Enviroment with ASM++
#Project_Name= Heat Sensor
#Creator= e1347061
#Created@ 2/12/2006 13:29:06
#Modified@ 2/12/2006 13:45:33
#Description= Ceng336 odevi icin yazdigimiz bir isi sensoru
```

Other Files in Project File

Project file holds trace of all files included in the project. These files may be ASM++ source files, ASM files, HEX files, debug files and test files. Each file is defined with its type and path name. The lines preceding types of the files begin with file type token (>) and file paths are saved after "FILE=" keyword. Below is an example of files.

```
>ASM++
FILE= ./source/heat sensor.asmpp
>ASMHEADER
FILE= ./myLib/a2dcalculate.ah
>ASMHEADER
FILE= ./d2acalculate.ah
>TESTFILE
FILE= ./testcase1.test
>DEBUGFILE
FILE= ./heat sensor.dbg
```

Workspace in ProjectFiles

Project file saves last snapshot of the workspace. When user opens an existing project, GUI will be modified according to these settings. This section begins with WORKSPACE_BEGIN keyword and ends with WORKSPACE_END keyword. Between the keywords states of all the views and windows are saved. View properties, i.e. visibility of toolbars, shortcuts, etc. are leaded with "VIEW_" tag and window properties, i.e. subwindows which were open just before leaving workspace, are leaded with "WINDOW_" tag. Editor windows are special cases since they require additional information like the file they are editing. There is an

editors section in the workspace between "WINDOW_EDITOR_LIST_BEGIN" keyword and "WINDOW_EDITOR_LIST_END" keyword. In this section a mode tag is followed by a file path. Below is an example of workspace.

```
WORKSPACE_BEGIN
VIEW_TOOLBAR_DEBUG= OFF
VIEW_BUTTON_DEBUG_STEP= ON
...
(removed)
...
WINDOW_EDITOR_LIST_BEGIN
FULL= NONE
FLOATING= ./source/heat sensor.asmpp
MINIMIZED= ./testcase1.test
WINDOW_EDITOR_LIST_END
...
removed
...
WINDOW_BUTTOM_CONSOLE= TABBED
WINDOW_BUTTOM_LOG= ON
WINDOW_SIDE_WATCHPOINT= TABBED
WINDOW_SIDE_REGISTERS= ON
WORKSPACE_END
```

7.3. Debug File Format

Debug files hold data of the source and binary executable files that will be used in debugging process. Debugger needs watchpoints and breakpoints to halt execution. Watchpoints are held as register addresses and breakpoints as line number of some source file. Debug file holds existing watchpoint and breakpoint locations in a file <project_name>.dbg. Below are the specifications and format of the debug file.

Cross Mappings of the Line Numbers for Breakpoints

Breakpoints are defined using source files. These lines should be mapped to corresponding lower level file lines. Breakpoints may be lying in different files so each files line number is separated from another. Breakpoint section begins with BREAKPOINT_BEGIN keyword and ends with BREAKPOINT_END keyword. After BREAKPOINT_BEGIN keyword, the path of the file to which source file line numbers are mapped is saved. This file is usually a generated asm file with file name <project_name>_g.asm. Each source file's breakpoint data is listed under its path name leaded with its source type. After each file, END_OF_BP_LIST keyword is used to indicate the source file has no other breakpoints. Each breakpoint is indicated with a >BP tag followed by line number of the associated source file and mapped line number. Other mappings simply don't have any tags. Below is an example of breakpoint section.

```
BREAKPOINT_BEGIN DEST= ./heat sensor_g.asm
ASMFILE= ./source/heat sensor.asmpp
4 1
5 2
...
...
11 11
>BP 12 14
13 16
...
...
45 50
>BP 46 55
...
...
81 90
END_OF_BP_LIST
ASMHEADER= ./myLib/a2dcalculate.ah
1 91
2 94
3 95
>BP 4 98
...
...
>BP 19 122
...
...
>BP 24 130
...
...
END_OF_BP_LIST
BREAKPOINT  END
```

Register Adresses for Watchpoints

Watchpoints are defined using registers of the microcontroller. They are mapped to a real address value in the PIC and debugger halts whenever a register referenced by a watchpoint is altered. Debugger receives line number information

to continue debugging process from simulator. Watchpoint section begins with WATCHPOINT_BEGIN keyword and ends with WATCHPOINT_END keyword. Each watchpoint is indicated with a >WP tag followed by register address of PIC in hexadecimal format. Some special registers are indicated with descriptive labels such as stack registers. Below is an example of watchpoint section.

```
WATCHPOINT_BEGIN
>WP 0x0101
>WP STACK1
>WP W
>WP STATUS
WATCHPOINT_END
```

7.4. ASM Header File Format

ASM header files (<file_name>.ah) are used to include predefined functions, procedures or macros. Content of an ASM header file is almost the same with a regular ASM++ file. User can include other ASM header files, define variables, functions and macros in an ASM header file. It is user's responsibility not to use the names used in header files he/she includes. Also user shouldn't include declarations for PIC's internal setup, i.e. setting watchdog off, which should be in an ASM++ file that contains the main program. PIDE supplies a set of header files that includes many procedures frequently used in embedded programming.

8. Coding Standarts

To increase maintainability of the source code, all project members will obey the coding standarts described below.

8.1. Coding Conventions

Inside the class scope, attributes and method declarations should be followed with method definitions. Attributes and method declarations shall be logically grouped using appropriate comments.
Class attributes should be private. All attributes must have its own getter and setter methods implemented.

8.2. Naming Conventions

Naming conventions will be as Java naming conventions.
Class names will be as descriptive as possible and initial letters of each word and abrreviation letters will be capitalized.

Example: Class, ClassName, CClass, ClassC etc.

For the class names we extended from Java libraries, class name will be preserved except that the 'J' letter at the beginnig is replaced with 'P'.

Example: class PMenuBar extends JMenuBar.

Method names and Class attributes always start with small letters. Each word or abrreviation letter after the first word or abbreviation letter will be start with capital letters.

Example: var, varP, varPoint, iPoint, varFirstSecond, varFS, method(), methodName(), mName(), methodN() etc.

Instances of classes have the same name with the class name, but the first letter will be lowercase.

Example: BreakPointHandler breakPointHandler = null;

8.3. Comments

Comment conventions will be as Java commenting conventions.
At the beginning of each file, there will be a descriptive comment which must include file name, creator, creation time, last edit date.

Classes, attributes and methods should be leaded with descriptive comments.

- Class comments should describe functionality of the class and may include special notes if any. The comment should have @author <author name> line in the end.
- Attribute comments should be brief as much as possible.
- Method comments should describe behavior and aim of the method. All parameters should be described using @param tag and return values should be described with @return. The comment should have @author <author name> line in the end. Local variables should be described inside the method.

8.4. Indentation

Indentation conventions will be as Eclipse Java Indentation conventions.
Scope defining curly braces should be put in a new line and indented to the same vertical line. Example:

```
Class Class1
{
        void method ()
        {
                if ( var1 == var2 )
                {
                        if ( var2 == var3 )
                        {
                                ...
                        }
                }
        }
}
```

To increase readibility, there should be white spaces before and after any names, operators, etc. Example:

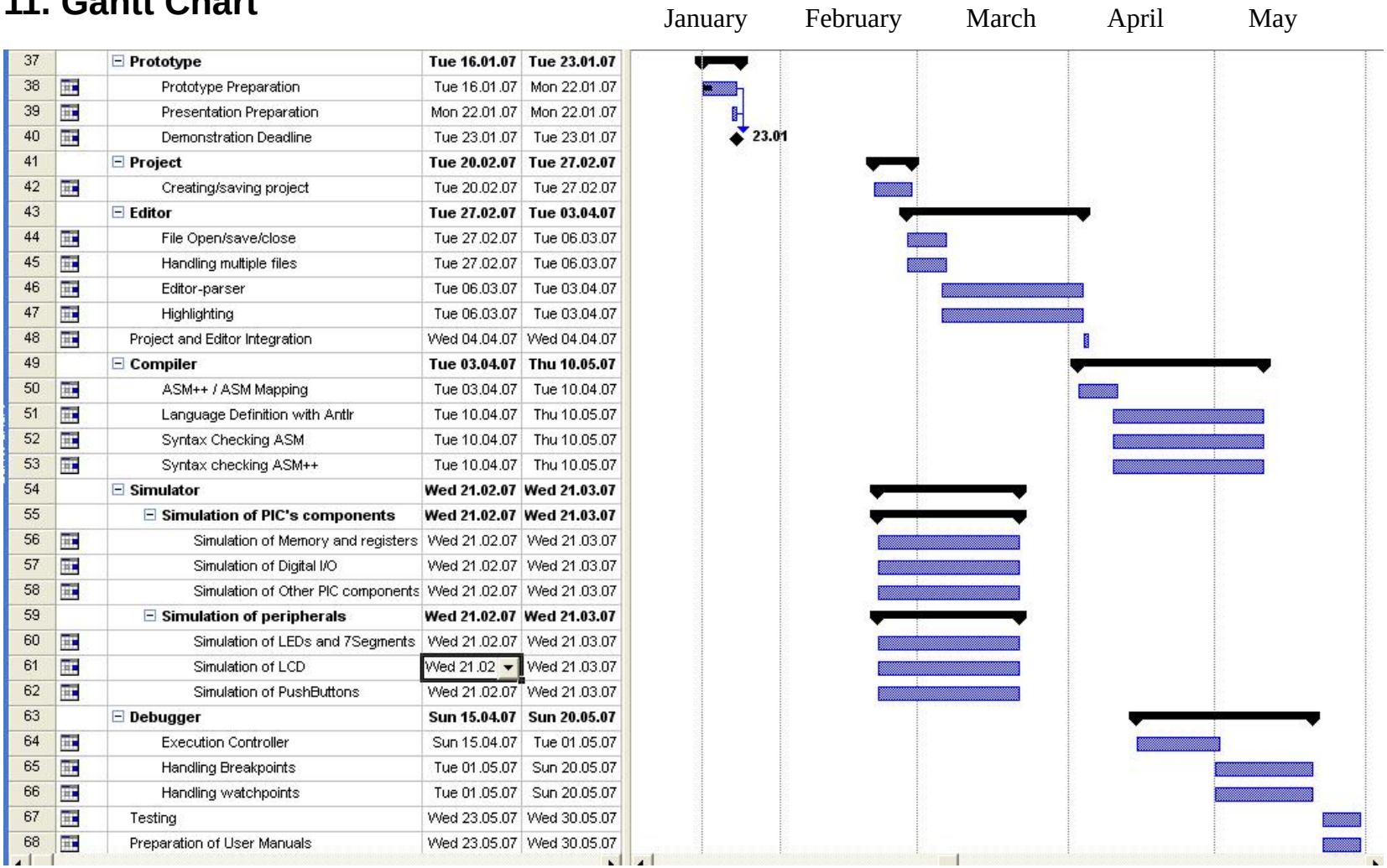
```
var = 3 + ( var1 + var2 * var3 / method() );
```

10. System Testing Considerations

To supply a faultless product, enough time should be given for testing. Before the demonstration of the project, one week will fully be dedicated for the testing of PIDE.
Since PIDE is being developed in an object oriented approach, object oriented testing strategies should be applied.

- Each class will be tested during the development time, to observe whether it operates correctly. White box testing will be applied at this level.
- Interclass tests will be applied to check the interactions between the classes.
- After completing each subsystem of the program, scenario based tests will be applied. For example, to compile an .asm file to test the compiler module, etc.
- After the completion of coding, the program will be tested to see whether it works correctly with full functionality.

11. Gantt Chart



37		**Prototype**	**Tue 16.01.07**	**Tue 23.01.07**
38		Prototype Preparation	Tue 16.01.07	Mon 22.01.07
39		Presentation Preparation	Mon 22.01.07	Mon 22.01.07
40		Demonstration Deadline	Tue 23.01.07	Tue 23.01.07
41		**Project**	**Tue 20.02.07**	**Tue 27.02.07**
42		Creating/saving project	Tue 20.02.07	Tue 27.02.07
43		**Editor**	**Tue 27.02.07**	**Tue 03.04.07**
44		File Open/save/close	Tue 27.02.07	Tue 06.03.07
45		Handling multiple files	Tue 27.02.07	Tue 06.03.07
46		Editor-parser	Tue 06.03.07	Tue 03.04.07
47		Highlighting	Tue 06.03.07	Tue 03.04.07
48		Project and Editor Integration	Wed 04.04.07	Wed 04.04.07
49		**Compiler**	**Tue 03.04.07**	**Thu 10.05.07**
50		ASM++ / ASM Mapping	Tue 03.04.07	Tue 10.04.07
51		Language Definition with Antlr	Tue 10.04.07	Thu 10.05.07
52		Syntax Checking ASM	Tue 10.04.07	Thu 10.05.07
53		Syntax checking ASM++	Tue 10.04.07	Thu 10.05.07
54		**Simulator**	**Wed 21.02.07**	**Wed 21.03.07**
55		**Simulation of PIC's components**	**Wed 21.02.07**	**Wed 21.03.07**
56		Simulation of Memory and registers	Wed 21.02.07	Wed 21.03.07
57		Simulation of Digital I/O	Wed 21.02.07	Wed 21.03.07
58		Simulation of Other PIC components	Wed 21.02.07	Wed 21.03.07
59		**Simulation of peripherals**	**Wed 21.02.07**	**Wed 21.03.07**
60		Simulation of LEDs and 7Segments	Wed 21.02.07	Wed 21.03.07
61		Simulation of LCD	Wed 21.02	Wed 21.03.07
62		Simulation of PushButtons	Wed 21.02.07	Wed 21.03.07
63		**Debugger**	**Sun 15.04.07**	**Sun 20.05.07**
64		Execution Controller	Sun 15.04.07	Tue 01.05.07
65		Handling Breakpoints	Tue 01.05.07	Sun 20.05.07
66		Handling watchpoints	Tue 01.05.07	Sun 20.05.07
67		Testing	Wed 23.05.07	Wed 30.05.07
68		Preparation of User Manuals	Wed 23.05.07	Wed 30.05.07