GWBasic64

High Level and Low Level Design Document

Tuesday 29th July, 2025

Document Control

Date	Version	Author	Brief Description of Changes	Approver Signature

Team Member - Module Assignment Table

Employee ID	**Name**	**Module Assigned**
293944	Remin Varghese	main / gwbasic64 / SystemInterface / ErrorHandler
293948	Vaishnavi	lexer / token
293961	Niviya	Runtime Execution / Expression Evaluation
294001	Magesh	Program Memory
293952	Vijaylaxmi	Program Interface/ CLI / special key handler
293943	Suvetha	Parser / ast node

Contents

List of Figures

Chapter 1

High Level Design

1.1 Introduction

1.1.1 Purpose

The purpose of this document is to describe the overall high-level design of the GWBasic64 interpreter system. This document serves as a blueprint for understanding how the various components interact, what tools and constraints guide the architecture, and how the software behaves under different operating environments, especially a freestanding custom OS.

1.1.2 Scope

GWBasic64 is a 64-bit interpreter for GW-BASIC programs. It supports both direct and indirect execution modes and runs in both standard operating systems (Windows/Linux) and freestanding environments (MiniOS). The interpreter supports parsing, interpreting, error handling, and file-based program execution using a modular architecture.

1.1.3 Definitions

- **Token** – A lexical unit such as ‘PRINT‘, ‘10‘, or ‘GOTO‘.
- **AST** – Abstract Syntax Tree built during parsing for program logic.
- **REPL** – Read-Eval-Print Loop for direct-mode command interaction.
- **StatementExecutor** – Executes parsed statements.
- **SystemInterface** – Abstraction over platform-dependent operations.

1.1.4 Overview

The remainder of this document covers general descriptions of tools and assumptions, followed by detailed design features, architecture, diagrams, and non-functional attributes like performance, reliability, and maintainability. All major classes and their roles are listed for easy reference.

1.2 General Description

1.2.1 Product Perspective

GWBasic64 is a standalone command-line interpreter that mimics legacy GW-BASIC behavior with a modern, modular architecture. It is intended to support integration with a minimal OS.

1.2.2 Tools Used

- **Languages:** C++17
- **Build System:** CMake (version 3.20+)
- **Testing Framework:** GoogleTest
- **Compilers:**
  - **MinGW (GCC):** for native Windows builds with C++17 support
  - **MSVC (Microsoft Visual C++):** via Visual Studio 2022 using '/std:c++17'
  - **x86_64-elf-g++ (Cross Compiler):** for building freestanding 64-bit ELF binaries targeting MiniOS
  - **g++ (GNU Compiler Collection):** for native Linux or WSL builds
- **IDE:** Visual Studio 2022

1.2.3 General Constraints

- Must run on both hosted and freestanding environments.
- Compiled to 64-bit ELF binaries only for MiniOS.

1.2.4 Assumptions

- Target processor follows x86-64 ISA.
- Programs are interpreted line-by-line.
- Single-core execution environment.
- All '.bas' files are valid.

1.2.5 Special Design Aspects

- Direct access to hardware I/O in MiniOS using syscall layer.
- Freestanding I/O handling modules.
- Modular system interface that can be overridden per platform.

1.3 Design Details

1.3.1 Main Design Features

- Dual-mode execution: direct (REPL) and indirect ('RUN filename.BAS')
- Full tokenization and AST construction for valid statements.
- Clean separation of Lexer, Parser, Executor, and System layers.

1.3.2 Application Architecture

- **Lexer** – Breaks input into tokens.
- **Parser** – Builds syntax tree from tokens.
- **ProgramMemory** – Stores and retrieves program lines.
- **StatementExecutor** – Executes each statement sequentially.
- **SystemInterface** – Handles platform-specific functions (I/O, files).
- **commandLineEditor** - Handles command line edit and special keys

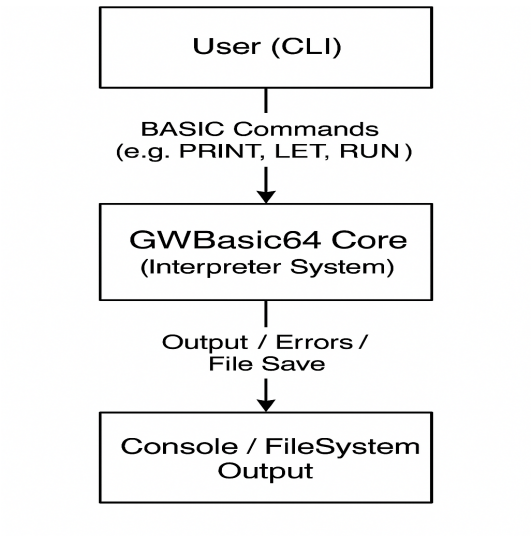

Figure 1.1: architectural diagram

1.3.3 Data Flow Diagram

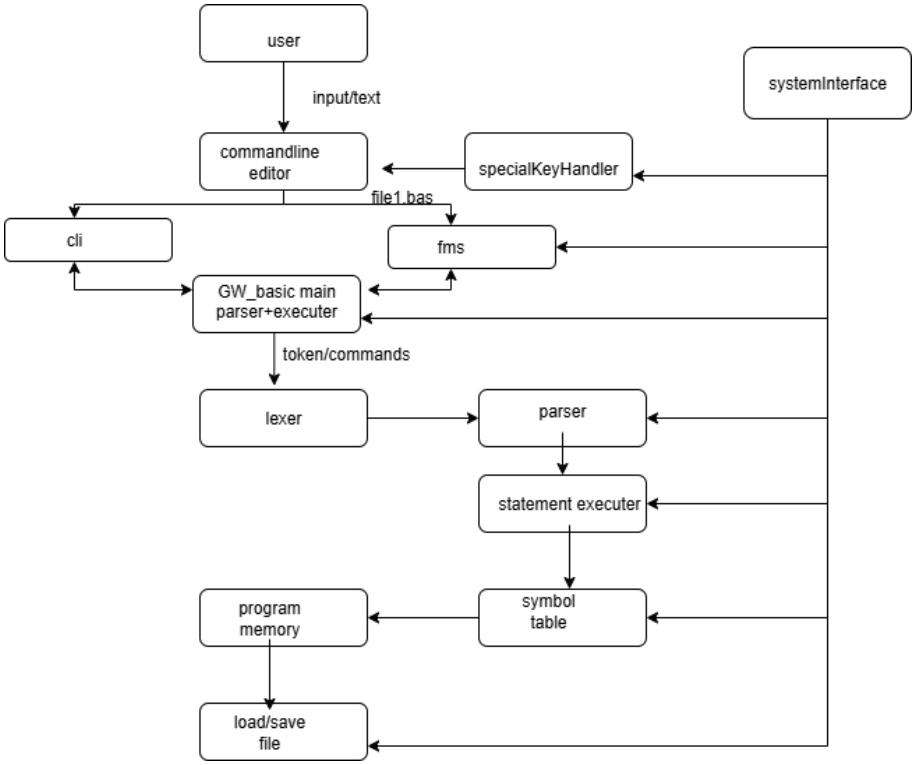

Figure 1.2: Data Flow diagram

1.3.4 Files

- `*.bas` – Source programs
- `*.cpp`, `*.h` – Interpreter source code
- `gwbasic64.exe` – Output for Windows
- `gwbasic64.elf` – Output for MiniOS

1.3.5 User Interface

- Command-line interface (REPL)
- Supports commands like 'RUN', 'LIST', 'EDIT', 'NEW' ,'SAVE' ,'LOAD' , 'AUTO' ,'RENUMBER' ,'EXIT' , 'CLS'
- Supports line-based program editing (e.g., '10 PRINT "HELLO"')

1.3.6 Error Handling

- Central 'ErrorHandler' class for consistent error messaging.
- Errors divided into: Syntax, Runtime, type ,system.

1.3.7 Portability

- Runs on Windows (need to implement- Linux, and custom OS).
- System interface abstracts away OS-specific APIs.

1.3.8 Application Compatibility

Supports legacy GW-BASIC syntax for numeric and string variables, control flow ('IF', 'GOTO', 'FOR'), and print statements.

Major Classes

- `GWBasic64` – Main interpreter entry point.
- `Lexer` – Tokenizes source input.
- `Parser` – Constructs AST from tokens.
- `StatementExecutor` – Executes parsed statements.
- `ProgramMemory` – Stores BASIC line numbers and code.
- `SystemInterface` – Manages I/O, file access
- `CommandLineEditor` – Handles command line edit buffer and special keys.

1.3.9 Major Classes

- GWBasic64
- Lexer
- Parser
- StatementExecutor
- ProgramMemory
- SystemInterface
- CommandLineEditor

Chapter 2

Low Level Design

2.1 Introduction

2.1.1 Purpose

2.1.2 References

- GW-BASIC Manual

2.2 Detailed System Design

2.2.1 Design Description

Detailed explanation of modules and submodules.

Main

Here the user gives enters either in direct mode or indirect mode which is directly loading a .bas file through command line arguments.

GWBasic64 - core module

The GWBasic64 Module serves as the central entry point for the GWBasic64 interpreter. It initializes all core subsystems (Lexer, Parser, Executor, Memory, cli editor and system interface I/O), processes user input in both direct and program modes, and orchestrates execution flow from command parsing to statement executionResponsibilities.

Responsibilities :

- Set up the interpreter environment.
- Initialize all core subsystems.
- Enter interactive REPL or load .BAS program file.
- Accept and process user commands (PRINT, LET, LIST, RUN, NEW, etc.).
- Handle errors and control flow (GOTO, IF, etc.).
- Manage shutdown sequence (e.g., EXIT commands)

Lexer

Module Purpose The **Lexer** module is responsible for performing *Lexical Analysis* on a single line of GW-BASIC source code. It reads the input line character by character and breaks it down into a sequence of tokens—the smallest meaningful units in the language (e.g., `PRINT`, `10`, `"HELLO"`, `+`, etc.). These tokens are passed to the parser in the next phase to understand the syntax and semantics of the code.

Core Responsibilities The Lexer performs the following key tasks:

- Recognizes and extracts:
  - **Line Numbers**
  - **Keywords** (e.g., `PRINT`, `GOTO`)
  - **Identifiers** (e.g., variable names like `A$`, `count1`)
  - **Numbers** (e.g., `100`, `3.14`, `1E5`)
  - **Strings** (e.g., `"Hello"`)
  - **Operators & Symbols** (e.g., `+`, `-`, `<=`, `<>`)
  - **Separators** (e.g., `,`, `:`)
  - **Comments** (e.g., `REM This is a comment`)
- Skips whitespace
- Appends an `END_OF_LINE` token to mark the logical end

Main Functionality **tokenize(const std::string& line)**:

- Initializes the line and character index
- Skips leading whitespace
- Extracts line number if present (via `extractNumber()`)
- Iterates through characters and identifies:
  - Numbers → `extractNumber()`
  - Keywords / Identifiers → `extractWordOrKeyword()`
  - Strings → `extractQuotedText()`
  - Operators / Symbols → `extractOperatorOrSymbol()`
- Appends `END_OF_LINE` token

Supporting Functions

- **skipSpaces()** – Skips whitespace characters (`' '`, `'\t'`, etc.)
- **getCurrentChar(), moveToNextChar()** – Utility functions for reading and advancing characters
- **extractNumber()** – Recognizes integers, decimals, and scientific notation (e.g., `1E3`)
  - Handles optional decimal point
  - Handles `E/e` for exponent
  - Handles `+/-` in exponent
- **extractWordOrKeyword()**
  - Reads alphanumeric words and dollar sign (e.g., `A$`, `B2`)
  - Converts word to uppercase for keyword matching
  - Detects and handles `REM` using `extractCommentAfterRem()`
- **extractQuotedText()** – Handles string literals inside `"..."`; supports escaped quotes like `\"`
- **extractOperatorOrSymbol()** – Identifies:
  - Operators: `+`, `-`, `*`, `=`, `>=`, `<=`, `<>`
  - Separators: `,`, `:`
  - Other unrecognized symbols
- **extractCommentAfterRem()** – Captures the entire remaining line as a comment token
- **isKeyword(std::string)** – Checks if a word is a GW-BASIC keyword
- **toUpper(std::string)** – Converts string to uppercase for case-insensitive matching

Example Input and Output **Input Line:**

```
10 PRINT "HELLO"
```

Output Tokens:

LineNumber: 10

Keyword: PRINT

String: "HELLO"

END_OF_LINE

Parser

Overview The `Parser` module converts a sequence of tokens (produced by the lexer) into an Abstract Syntax Tree (AST). This tree represents the syntactic structure of a GW-BASIC program and forms the basis for interpretation or compilation.

Goals

- Implement a recursive-descent parser for a subset of GW-BASIC.
- Support common statements such as `LET`, `PRINT`, `IF/THEN`, `FOR/NEXT`, `WHILE/WEND`, `DO/LOOP`, `GOSUB`, `RETURN`, etc.
- Build a well-structured and memory-safe AST hierarchy.
- Provide descriptive error handling for invalid syntax.

Architecture

Key Components

- **Parser**: Entry point for all parsing logic.
- **ASTNode.h**: Contains all AST node classes.
- **Token.h**: Token structure used for parsing.
- **parser.cpp**: Implements recursive-descent parsing methods.

Parser Flow

Lexer Output → [Tokens] → Parser → [AST Root (`ProgramNode`)]

Parsing Strategy Recursive-descent parsing is used, where each grammar rule corresponds to a dedicated C++ function:

- `parseProgram()` — Parses a full program.
- `parseStatement()` — Parses a single statement.
- `parseExpression()` — Parses arithmetic/comparison expressions.
- `parseTerm()` and `parseFactor()` — Handle lower-precedence grammar rules.

AST Structure

Base Class

```
class ASTNode {
public:
    virtual ~ASTNode() {}
    virtual ASTType type() const = 0;
};
```

Statement Nodes

- `ProgramNode`: Holds a list of statements.
- `LetNode`, `PrintNode`, `InputNode`, `IfElseNode`, `ForNode`, `NextNode`, `GotoNode`, `ReturnNode`, `StopNode`, `RemNode`, etc.
- `CommandNode`: Generic handler for shell commands (`SYSTEM`, `DIR`, `CONT`, etc.)

Expression Nodes

- `BinOpNode`: Binary operations (+, -, *, /, ¡, ¿, etc.).
- `NumberNode`, `StringNode`, `IdentNode`.
- `MathFuncNode`: Built-in math functions (e.g., `SIN`, `COS`, `LOG`).

Token Handling

- `peek()` — Returns the current token without consuming it.
- `get()` — Consumes and returns the next token.
- `match()` — Checks and consumes a token based on type/value.

Parsing Functions

parseProgram()

- Skips line numbers and parses statements until the end-of-line.
- Handles multiple statements separated by `:`.

parseStatement()

- Dispatches based on the current keyword. Example:

```
if (match(TokenType::Keyword, "PRINT")) return parsePrint();
if (match(TokenType::Keyword, "LET"))   return parseLet();
if (match(TokenType::Keyword, "IF"))    return parseIf();
```

- Supports implicit assignments (e.g., `X = 5`).

parseExpression()

- Builds expression trees for binary comparisons (`<`, `>`, `=`).
- Delegates to `parseTerm()` and `parseFactor()` for operator precedence.
- Parses math function calls like `SIN(x)` using `parseMathFunc()`.

Error Handling

- Throws `std::runtime_error` on invalid syntax.
- Custom messages identify the source of the failure.
- Uses synchronization techniques (e.g., skip tokens until `:` or `END_OF_LINE`) for recovery.

Sample Grammar (Subset)

```
program    ::= { line }
line       ::= [Number] statement { ':' statement }
statement  ::= LET | PRINT | IF | FOR | GOSUB | ...
expression ::= term { ('+' | '-' | '<' | '>' | '=') term }
term       ::= factor { ('*' | '/') factor }
factor     ::= NUMBER | IDENTIFIER | STRING | '(' expression ')' | mathFunc
```

Supported Statements The parser supports a wide range of GW-BASIC statements:

- `LET`, `PRINT`, `INPUT`
- `IF/THEN/ELSE`, `FOR/NEXT`, `WHILE/WEND`, `DO/LOOP`
- `GOSUB/RETURN`, `STOP/END`, `DATA/READ`, `REM`
- `ON ERROR GOTO`, `FIELD`
- Shell commands: `SYSTEM`, `CONT`, `EDIT`, `CLEAR`, `DIR`, `SCREEN`, etc.

Extensibility

- Add a semantic analysis phase after AST generation.
- Integrate an interpreter or bytecode generator.
- Expand AST types for user-defined functions and subroutines.

Conclusion The `Parser` module forms the backbone of the GW-BASIC interpreter/compiler. It cleanly separates syntax analysis from semantics and provides detailed error diagnostics. It is designed to be robust, extensible, and maintainable, making it suitable for both educational and hobbyist interpreter development.

Runtime

The Runtime module forms the core execution engine of the GW-BASIC 64 interpreter. It handles variable storage, type management, flow control, expression evaluation, and the execution of all statements parsed from the Abstract Syntax Tree (AST).

Runtime Modules

1. **TypeSystem**
   Purpose: Manages typed values used in expressions and variable assignments.
   Key Class: `Value`

   - **Features Implemented**:
     - Supports `int`, `float`, and `string` types.
   - **Methods**:
     - `asInt()`, `asFloat()`, `asString()` — Access values as a specific type.

- **Used By**: `ExpressionEvaluator`, `FlowControl`, `SymbolTable`

2. **SymbolTable**
   Purpose: Stores variables and their values.
   Key Class: `SymbolTable`

   - **Features Implemented**:
     - `setVariable(name, value)` — Assign a value to a variable.
     - `getVariable(name)` — Retrieve a variable's value; throws error if undefined.
   - **Used By**: All statements involving variables (`LET`, `INPUT`, `READ`, `FOR`).

3. **FlowControl**
   Purpose: Evaluates logical conditions for control-flow statements.
   Key Class: `FlowControl`

   - **Features Implemented**:
     - Supports comparisons: `=`, `<`, `>`, `<=`, `>=`, `<>`
     - `evaluateCondition(op, leftValue, rightValue)` — Evaluates a logical condition.
   - **Used By**: `IF`, `IF-ELSE`, `FOR`, conditional `GOTO`

4. **IOHandler**
   Purpose: Manages user input during program execution.
   Key Class: `IOHandler`

   - **Features Implemented**:
     - `getInput(prompt)` — Displays a prompt and reads input from the user.
   - **Used By**: `INPUT` statement.

5. **DataManager**
   Purpose: Manages `DATA` and `READ` statements.
   Key Class: `DataManager`

   - **Features Implemented**:
     - `addData(vector<Value>)` — Stores `DATA` values.
     - `readNext()` — Returns the next unread value.
     - `reset()` — Resets the read pointer.
   - **Used By**: `DATA`, `READ`

6. **ExpressionEvaluator**
   Purpose: Evaluates expressions and returns typed results.
   Key Class: `ExpressionEvaluator`

   - **Features Implemented**:
     - Supports expression types:

    - `NumberExpr` — Numeric constants
    - `StringExpr` — String literals
    - `IdentExpr` — Variable references
    - `BinOpExpr` — Binary operations (`+`, `-`, `*`, `/`)
  - **Used By**: `LET`, `PRINT`, `IF`, `FOR`, `INPUT`, `READ`

7. **StatementExecutor**
   Purpose: Executes all program statements from the AST.
   Key Class: `StatementExecutor`

   - **Features Implemented**:
     - **Statement Types Supported:**
       - `PRINT` — Displays strings or evaluated expressions.
       - `LET` — Assigns value to a variable.
       - `IF THEN` — Executes a statement conditionally.
       - `IF THEN ELSE` — Executes one of two branches.
       - `GOTO` — Jumps to a labeled line number.
       - `FOR / NEXT` — Implements counting loops.
       - `INPUT` — Prompts user and stores input in a variable.
       - `DATA` — Declares constant data values.
       - `READ` — Loads `DATA` values into variables.
       - `STOP` — Ends program execution.
       - `REM` — Comment line (ignored).

Total Language Features Implemented

- **Statements**: `PRINT`, `LET`, `IF`, `IF-ELSE`, `FOR`, `NEXT`, `GOTO`, `INPUT`, `DATA`, `READ`, `STOP`, `REM`
- **Expressions**: `NumberExpr`, `StringExpr`, `IdentExpr`, `BinOpExpr`
- **Execution**: Orchestrated by `StatementExecutor` with integration of all runtime modules.

Program Interface

Introduction The `ProgramInterface` module provides the user interface layer for the GW-BASIC-style interpreter. It enables live command-line editing, special key handling (e.g., function keys), and console rendering for a fully interactive text-based environment. The module ensures that users can efficiently type, edit, and execute BASIC-like commands via keyboard input.

Dependencies This module depends on the following components:

- **SystemInterface**: Provides platform-independent input/output functions (keyboard and screen).
- **ScreenRenderer**: Manages cursor movement and buffer redrawing on the console.
- **SpecialKeyHandler**: Maps function keys (F1–F10) to shortcut commands (e.g., `RUN`, `LIST`).
- **check_SKey Structure**: Encapsulates both normal and special key input.

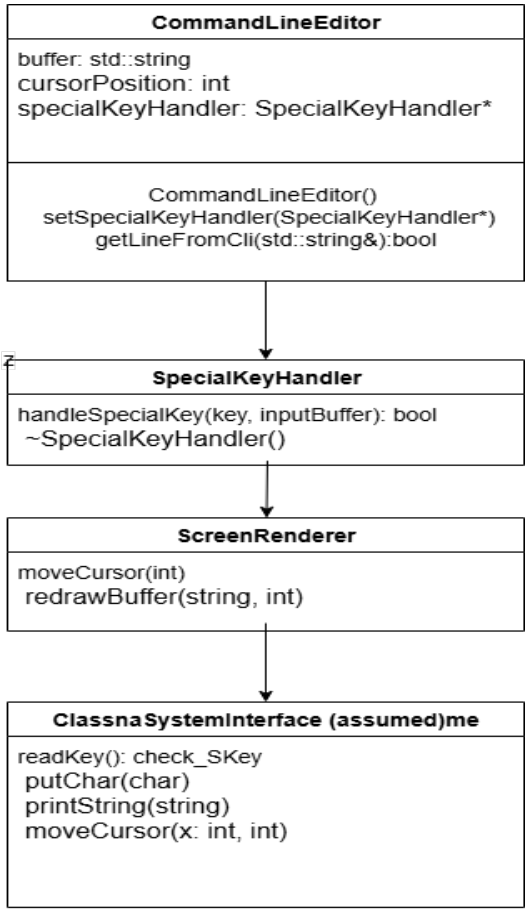

Figure 2.1: Classes defined

Class Descriptions

- **CommandLineEditor**
  Handles line-based text input from the user. Supports editing operations (insertion, deletion), cursor navigation, and special key interpretation (e.g., ENTER, ESC, arrow keys).
- **SpecialKeyHandler**
  Captures function key input (F1–F10) and maps them to predefined commands like `RUN` or `LIST`. Injects the mapped command into the input buffer.
- **ScreenRenderer**
  Provides methods to render the input buffer on-screen and control cursor position, enabling a classic DOS-style interface.

Public Methods CommandLineEditor

- `CommandLineEditor()` — Constructor to initialize the editor.
- `void setSpecialKeyHandler(SpecialKeyHandler* handler)` — Links a handler to interpret special keys.
- `bool getLineFromCli(std::string& line)` — Captures a complete line of user input with editing and special key support.

SpecialKeyHandler

- `bool handleSpecialKey(const check_SKey& key, std::string& inputBuffer)` — Maps function keys to their respective command strings and injects them into the editor buffer.

ScreenRenderer

- `static void moveCursor(int pos)` — Moves the cursor to a specific position within the input line.
- `static void redrawBuffer(const std::string& buffer, int cursorPos)` — Repaints the current input buffer on the screen with updated text and cursor position.

Internal Data Members CommandLineEditor

- `std::string buffer` — Holds the current input text.
- `int cursorPosition` — Tracks the cursor's index in the buffer.
- `SpecialKeyHandler* specialKeyHandler` — Reference to the function key handler.

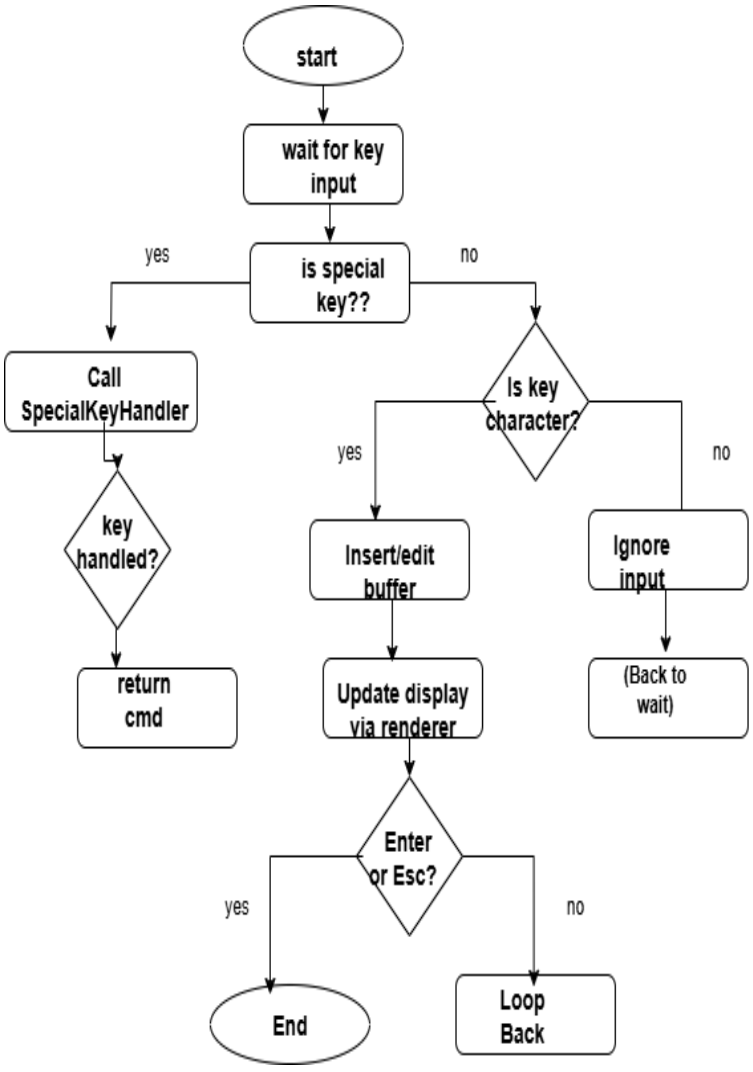

Figure 2.2: data flow

Storage

Overview The `ProgramMemory` module is designed to store and manage lines of code associated with specific line numbers, similar to how BASIC programs are structured. It uses a `std::map` to maintain the order and quick access of lines, enabling features such as adding, updating, deleting, listing, and renumbering lines. The module also supports saving and loading program lines from files using the `SystemInterface`, providing persistent storage across sessions.

Public Methods Overview

- `ProgramMemory()`
  Description: Constructor that initializes an empty program memory.
- `void storeLine(int lineNumber, const std::string& code)`
  Description: Stores or updates a line of code in memory. If the code is empty, the line is deleted. **Throws**: `std::invalid_argument` for negative line numbers.
- `void storeLine(const std::string& line)`
  Description: Parses a full input string like `"10 PRINT "HELLO"` and stores it after separating the line number and code. **Throws**: `std::invalid_argument` if line number is missing or invalid.
- `std::map<int, std::string> getAllLines() const`
  Description: Returns a copy of the internal memory map.
- `void list() const`
  Description: Prints all stored lines to the system using `SystemInterface::printString`.
- `std::vector<std::string> getLineNumbers() const`
  Description: Returns a list of all line numbers as strings.
- `void clearMemory()`
  Description: Clears the entire program memory.
- `bool saveToFile(const std::string& path) const`
  Description: Saves all stored program lines to a file. **Uses**: `SystemInterface::createAndSaveFile`
- `bool loadFromFile(const std::string& path)`
  Description: Loads program lines from a file, replacing current memory. **Uses**: `SystemInterface::oper`
  `SystemInterface::readLineFromFile()`, `SystemInterface::closeFile()` Returns `false`
  if the file cannot be opened.
- `bool lineExist(int lineNumber) const`
  Description: Checks if a specific line number exists in memory.
- `void deleteLine(int lineNumber)`
  Description: Deletes a line with the given line number from memory.
- `std::string getLine(int lineNumber) const`
  Description: Returns the code corresponding to the given line number (or empty string if not found).
- `void renumber(int newStart, int oldStart, int increment)`
  Description: Renumbers all lines starting from `oldStart`, assigning new numbers starting at `newStart` with increment step. Lines below `oldStart` remain unchanged.

- `int getFirstLineNumber() const`
  Description: Returns the smallest line number in memory or `-1` if memory is empty.
- `int getNextLineNumber(int current) const`
  Description: Returns the next higher line number after `current`, or `-1` if none exists.

Private Members

- `std::map<int, std::string> memory`
  Description: Stores the program lines in a sorted map, automatically ordered by line number.

Conclusion The `ProgramMemory` module efficiently manages program lines using line numbers. It supports adding, deleting, listing, renumbering, and performing file operations, making it a critical part of any BASIC-like interpreter or REPL system. Its design ensures simple and organized storage of user programs.

Error Handler

Module Purpose
The Error Handler module centralizes the reporting of different types of errors that may occur during interpretation of GW-BASIC programs. It provides user-friendly and context-aware messages to help debug issues during lexical analysis, parsing, and execution. The module outputs errors via the SystemInterface, allowing consistent feedback to the user.

Core Responsibilities

- Detect and report syntax errors (e.g., malformed expressions or keywords).
- Detect and report runtime errors (e.g., invalid logic such as GOTO to undefined line).
- Detect and report type or domain errors (e.g., division by zero, invalid string operations).
- Report critical system-level errors (e.g., file I/O failures).
- Differentiate between errors in *direct mode* and errors occurring in stored lines with line numbers.

Key Functions and Their Roles

- `runtimeError(int lineNo, const std::string& msg)`:
  Reports errors that occur during program execution. Indicates whether the error occurred in direct mode or on a specific line.
- `syntaxError(int lineNo, const std::string& msg)`:
  Reports structural or grammatical issues in the source code, such as invalid expressions or incomplete statements.
- `typeError(int lineNo, const std::string& msg)`:
  Used for semantic or domain-specific issues, such as mathematical errors like dividing by zero or incompatible data types.
- `systemError(const std::string& msg)`:
  Handles system-level exceptions like file I/O errors or unexpected runtime conditions. These errors are not tied to a specific line.

Integration with Interpreter Flow

- In `executeProgram()`:
  - `syntaxError` is invoked for invalid syntax via `std::invalid_argument`.
  - `runtimeError` is triggered for logic issues using `std::logic_error`.
  - `typeError` is used for domain-related errors (`std::domain_error`).
  - `systemError` handles unexpected exceptions like file read errors.
- In `runREPL()`:
  - Direct mode errors (no line number) are reported with lineNo = -1.
  - `executeLine()` uses the same error handler for ad hoc command evaluation.

Design Considerations

- Supports localization and redirection of error output via `SystemInterface::printString()`.
- Ensures graceful exit or continuation based on type and severity of error.
- Uses consistent formatting for error messages:
  `[Error Type] line <lineNo> : <error message>` or `[Error Type] (directMode): <error message>`

Example Output

- `[Syntax Error] line 30 : Missing closing quote in string literal`
- `[Runtime Error] line 100 : GOTO to undefined line number`
- `[Type Error/Domain Error] line 50 : Division by zero`
- `[System Error] : Unable to open file: test.bas`
- `[Syntax Error] (directMode): Unexpected token after PRINT`

System Interface

Module Purpose

The `SystemInterface` module abstracts low-level system operations such as keyboard input, screen control, and file I/O, enabling a hardware-independent interaction layer for the GW-BASIC 64 interpreter. It provides utility functions for reading special keys, printing characters, controlling the cursor, managing files, and rendering interface elements like the status bar.

Responsibilities

- Handle character and special key input from the keyboard.
- Provide output functionality for text and individual characters.
- Manage screen operations like clearing the screen and moving the cursor.
- Draw a function key status bar at the bottom of the screen.
- Support file I/O operations for reading and saving programs.

Key Data Structures

- `enum class SpecialKey`: Enumerates keys like arrow keys, function keys (F1–F10), ESC, CTRL-Z, and CTRL-C.
- `struct check_SKey`: Represents the result of a keyboard input, including whether the key is special and its type or character.

Public Methods

- `static void init()`
  Initializes any necessary system-level resources (currently empty placeholder).
- `static check_SKey readKey()`
  Reads a key press from the user, including support for special keys and standard characters.
- `static void putChar(char c)`
  Outputs a single character to the console.
- `static char getChar()`
  Reads a single character input using low-level keyboard access.
- `static void printString(const char* str)`
  Prints a string to the console and flushes the output stream.
- `static void clearScreen()`
  Clears the screen using ANSI escape codes.
- `static void moveCursor(int x, int y)`
  Moves the cursor to a specified position on the terminal using ANSI escape codes.
- `static void drawStatusBar()`
  Displays a fixed bottom-line status bar showing function key mappings (F1–F10).
- `static bool openFile(const std::string& filename)`
  Opens a file for reading and stores the handle internally.
- `static void closeFile()`
  Closes the currently open file if it is open.
- `static bool readLineFromFile(std::string& line)`
  Reads a single line from the open file into a string. Returns false on EOF.
- `static void createAndSaveFile(const std::string& path, const std::vector<std::string>& lines)`
  Creates a file at the given path and writes all lines of code into it. Throws an error if the file cannot be opened.

Design Considerations

- Separates special keys (function keys, arrows) from regular characters using a struct.
- Uses ANSI escape sequences to control terminal screen and cursor behavior.
- Maintains compatibility with DOS-style key input through `_getch()`.
- Ensures cross-module reusability for printing, input handling, and file interaction.

- Provides flush-after-print behavior for immediate display of output.

Example Use Cases

- Interactively reading input during REPL or command line mode.
- Rendering output messages and error logs.
- Navigating or redrawing the screen after program edits.
- Saving/loading user-written programs from and to files.

2.2.2 Class Diagram

UML class diagram to show core classes and relations.

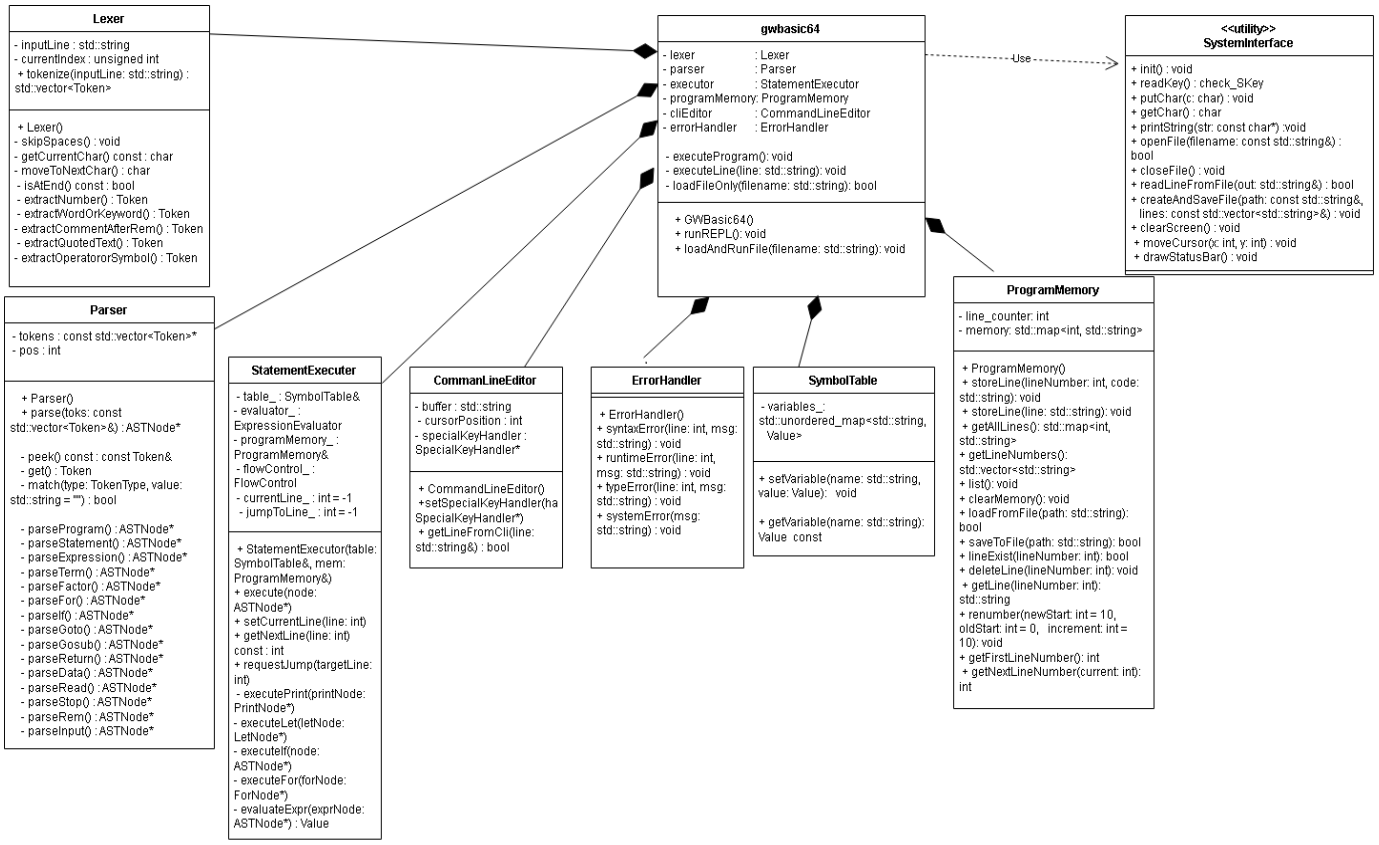

Figure 2.3: class diagram

2.2.3 Use Case Diagram

Show REPL commands, file loading, line execution.

2.2.4 User Interface

The user interface of the GW-BASIC 64 interpreter operates in a classic text-based console mode, offering two main interaction styles: **REPL (Read-Eval-Print Loop)** and **program mode input**. This design closely resembles traditional BASIC interpreters, enabling both interactive and stored program execution.

Text-Mode REPL The REPL mode allows users to directly type commands or statements and receive immediate feedback:

- The interface displays a prompt symbol `>` on each iteration, followed by user input.
- Special commands such as `LIST`, `RUN`, `NEW`, `SAVE`, `LOAD`, `DELETE`, and `EXIT` are recognized and executed immediately.
- Commands entered without a line number are executed in *direct mode*, meaning they are evaluated immediately without being stored in memory.

- Blank lines or pressing `Enter` twice display the message `Ok`, signifying that the interpreter is ready for the next input.
- Additional features like `AUTO` (for automatic line numbering) and `RENUMBER` (to renumber program lines) enhance productivity.

In REPL mode, input handling supports live command editing, cursor navigation, and special key detection (via the `CommandLineEditor` and `SystemInterface`). The status bar at the bottom of the console displays function key mappings (F1–F10) to streamline operations.

Program Mode Input Program mode allows users to enter and store multiple lines of a program, each associated with a line number:

- When input starts with a line number (e.g., `10 PRINT "HELLO"`), the line is stored in the `ProgramMemory` module instead of being executed immediately.
- Existing lines with the same line number are replaced, and empty code after the line number deletes the line.
- Programs can be listed using `LIST`, modified using `EDIT`, and cleared entirely using `NEW`.
- Once a program is fully entered, the `RUN` command executes the stored lines sequentially.
- Lines can also be renumbered for better readability using the `RENUMBER` command.

Program mode input is persistent and can be saved or loaded from files:

- `SAVE "filename.bas"` stores all program lines to disk.
- `LOAD "filename.bas"` loads a program file into memory, replacing any existing lines.

End of Document