Linking Loader Programmer’s Guide

Lab 3 Group BEERZ

**Evan Todd,**

**Brad Kline, Elliot Schumacher, Ryan Powers, Zach Smith**

**3/18/2011**

Contents

[Introduction 3](#_Toc287593071)

[Linking Loader Overview 3](#_Toc287593072)

[Directory Structure 3](#_Toc287593073)

[Design Conventions 4](#_Toc287593074)

[Module Inter-Relationships 4](#_Toc287593075)

[Data Structures 4](#_Toc287593076)

[Component Descriptions 6](#_Toc287593077)

[Client-Side 6](#_Toc287593078)

[Implementer-Side 6](#_Toc287593079)

[Main 6](#_Toc287593080)

[Loader 8](#_Toc287593081)

[ByteOperations 9](#_Toc287593082)

[ObjectFile 10](#_Toc287593083)

[Error 12](#_Toc287593084)

[SymbolEntry 12](#_Toc287593085)

[MemoryBank 13](#_Toc287593086)

[Symbol 15](#_Toc287593087)

[SymbolTable 16](#_Toc287593088)

# Introduction

This document contains the summary of the implementation of the “Wi-11 Machine” linking loader, and will cover all of the components that are used in the execution of the linking loader. The linking loader will accept one or more object files that have been assembled, properly combine the object files and produce a single object file that is capable of being run on the “Wi-11 Machine” simulator. Object files are to follow the format laid out in the Assembler User’s Guide; this document is written cased on the assumption that the reader is familiar with this format. Throughout the guide, names of classes and system defined types will be in **Bold Courier Font**. Classes and methods are public unless stated otherwise.

# Linking Loader Overview

The linking loader is capable of handling multiple object files, and linking them together in order to produce a single executable file that can be input to the simulator. The linking loader is able to handle linking multiple relocatable programs, one relocatable program with no external symbols, and one absolute program with no external symbols. When the linking loader runs, the entry point is the class entitled **Main**, which is implemented in the **Loader** package in the **Main.java** file. **Main** is responsible for accepting and handling command line arguments and producing the simulator-executable object file.

## Directory Structure

The loader's files are in a parent directory named “cse-560-beerz,” that contains two subfolders named Loader and Common (the Common subfolder contains classes that are used in all three components of the Wi-11 machine). The following files are in the Loader directory:

1. **Main.java**
2. Contains the implementation of the **Main** class, which handles all disk and console I/O and oversees the linking process.
3. **Loader.java**
4. Contains the implementation of the stateless **Loader** class, which parses object files into instances of the **ObjectFile** class.
5. **ObjectFile.java**
6. Implements the **ObjectFile** class, which represents an object file loaded in memory. This class stores an object file's segment name, execution start address, text records, imported and exported symbols, and relocation information.

The following files are in the Common directory:

1. **ByteOperations.java**
2. Implements the static **ByteOperations** utility class, which provides functions for modifying binary data.
3. **Error.java**
4. The **Error** class is used by the Assembler, Loader, and Simulator to keep track of errors encountered while assembling, loading, or simulating a program.
5. **MemoryBank.java**
6. A **MemoryBank** represents an array of 16-bit memory. It provides functions for reading and writing memory, as well as resolving symbol values and relocating binary data.
7. **SymbolEntry.java**
8. A **SymbolEntry** represents a location in memory that must be modified by the **Loader**. The **SymbolEntry** may be associated with a **Symbol**, in which case the **Symbol**'s value must be inserted at the memory location. Otherwise, the **SymbolEntry** just indicates that the value in memory must be modified if it is relocated.
9. **Symbol.java**
10. A **Symbol** is a name/value pair with an extra flag that defines whether the **Symbol** is relocatable or constant.
11. **SymbolTable.java**
12. A **SymbolTable** is an internal representation mapping names to **Symbols**.

The Loader folder contains the subfolder “/tests” that contains the test assembly and object files that were used in the testing of the Loader.

The Loader folder also contains the subfolder “/docs” that contains the User’s Guide, Programmer’s Guide and the Testing Plan. Contained within the “/docs” directory is another subfolder “/Meeting Minutes” that contains the minutes for each design meeting that took place.

## Design Conventions

Across the files in our implementation, several design conventions were followed. These include the capitalization of class names and package names, like **Main** or **Loader**, lower-case names for variables, and mixed-case names for method titles like “getSegmentName”. Also, care was taken to limit the length of lines to less than 90 characters. On a whole, our lab was designed to be object-oriented. The objects that are used to represent the symbols and object files follow the object-oriented convention of encapsulation and data hiding.

## Module Inter-Relationships

The core linking loader algorithm is implemented in the **Main** and **Loader** classes. These modules use the **ObjectFile**, **SymbolTable**, **Symbol**, **SymbolEntry**, and **MemoryBank** classes to represent the state of the object files. In addition, the **Error** class is used for error reporting, and the **ByteOperations** utility class is used for binary modification functions.

# Data Structures

There are several important data structures that are used by the loader in the creation of the output executable. The following is a listing of the data structure, the file it appears in, the usage of the data structure, its implementation, and its invariant.

1. Object: “ObjectFile” defined by ObjectFile.java
2. Represents a complete “Wi-11 Machine” object file, including the segment name, execution start address, text records, imported and exported symbols, and relocation information.
3. Implemented as a **String** for the segment name, an **int** for the start address, a **MemoryBank** for the text records, two **List**s of **SymbolEntry**s for the imported and exported symbols and the relocation information.
4. Invariant: length of the segment name must be 6 and the start address must be in the range [x0, xffff]. The **MemoryBank** must not be **null**.
5. Object: “MemoryBank” defined by MemoryBank.java
6. Represents an array of 16-bit memory.
7. Implemented as a **HashMap<Integer, Short>** and two **int**s to track the first and last address containing data.
8. Invariant: the first and last addresses tracked by the **MemoryBank** must correspond to the smallest and largest **int** keys (respectively) in the **HashMap<Integer, Short>**.
9. Object: “SymbolEntry” defined by SymbolEntry.java
10. Represents a location in memory that must be modified by the **Loader**.
11. Implemented as a String that may be **null** for relocation records, or may contain the name of a **Symbol** associated with the entry, and three **int**s which represent a memory address and the indices of the most significant and least significant bits that need modified.
12. Invariant: the address must be in the range [x0, xffff], the index of the most significant bit must be larger than that of the least significant bit. Both bit indices must be in the range [0, 16].
13. Object: “Error” defined by Error.java
14. Represents a loader error with a message and an optional object file line number.
15. Implemented as a **String** to store the message, an **int** to store the line number, and a **boolean** to indicate whether the error makes use of the line number.
16. Invariant: the message must not be **null**, and the line number must be greater than zero.
17. Object: “Symbol” defined by Symbol.java
18. Represents the name/value pair of a defined symbol.
19. Implemented as a **String** for the symbol name, an **int** to store its value, and three **boolean**s indicating whether the symbol is relocatable, imported, or exported.
20. Object: “SymbolTable” defined by SymbolTable.java
21. Represents the internal mappings of names to **Symbols**.
22. Implemented as a **HashMap<String, Symbol>**.
23. Invariant: Only unique **Symbols** can be entered into the **SymbolTable**.

# Component Descriptions

This section provides a detailed description of both the client- and implementation-side view of every component used in the program.

## Client-Side

1. **Main**
2. Description: The **Main** component is the entry point into the loader program. It accepts the command line arguments from the user, such as the input files, output filename, and an option to generate a listing.
3. **Loader**
4. Description: The **Loader** component parses object files into in-memory **ObjectFile** reperesentations, which can then be relocated and linked.
5. Mathematical Model:

The Loader has no state.

1. Constraint:
2. **ObjectFile**
3. Description: The **ObjectFile** component holds the in-memory representation of an object file. The **ObjectFile** can then be relocated and linked into an executable file.
4. Mathematical Model:

ObjectFile = Start Address + Segment Name + MemoryBank + Symbol Entries + Relocation Records + Relocatable

## Implementer-Side

### Main

1. Description: The **Main** class contains the main method which is the entry point into the loader program.
2. State: None
3. Algorithm:

Parse input filenames from arguments

Parse output filename from arguments and attempt to open an output stream

Determine whether to generate a listing from the arguments

Parse the load address from arguments, if one is given

Read all input file data into Strings

Parse each input file into an **ObjectFile** and relocate appropriately

Resolve symbol values in each **ObjectFile** and combine them into one **MemoryBank**

Write header record, text records, and end record

Close output stream

1. **Main Methods**
2. Method Name: **main**
3. Description: Program entry point
4. Parameters: args[]
5. Requires: 1 or more input files and an output filename
6. Alters: outfile
7. Returns: void
8. Throws: None
9. Method Name: printUsageInformation
10. Description: Prints usage information to the console. Shows:

Usage: java Loader.Main [inputfiles] -o outfile [options]

-l Generating listing

-a addr Relocate program to addr (4-digit hex memory address)

Note: if linking an absolute object file, it must come first in the file list.

1. Parameters: None
2. Alters: System.out
3. Ensures: System.out = #System.out + [usage info]
4. Returns: void
5. Method Name: readAllText
6. Description: Reads all text in the file at the given path location into a string.
7. Parameters: **String** filename - Path to desired file
8. Requires: valid file
9. Returns: A string containing all the data existing in the desired file.
10. Throws: IOException
11. Method Name: writeAllText
12. Description: Writes all the given text to the file existing at the path location
13. Parameters: **String** filename - Desired output file name, **String** data – data to be written to file
14. Alters: The specified file on the file system
15. Ensures: The specified file is overwritten or created if it does not exist on the file system
16. Returns: void
17. Throws: IOException

### Loader

1. Description: The **Loader** parses an object file into an **ObjectFile** representation.
2. State: none
3. Alogrithm:

for each line in object file:

if line is a header record:

Extract the segment name, load address, and segment size

else if line is a text record:

Extract address and value and write them to the output **MemoryBank**.

if line has a modification record suffix:

Create and save a **SymbolEntry** for the address.

else if line is an export record:

Extract the symbol name and value

Define a new **Symbol** in the output **SymbolTable**

else if line is an import record:

Extract the symbol name, address, and bit mask

Create a new **SymbolEntry** with the name, address, and bit mask

else if line is an end record:

Extract the execution start address

end for

Create and return a new **ObjectFile** with the parsed segment name, load address, **MemoryBank**, **SymbolTable**, and lists of **SymbolEntry**s.

1. **Loader** **Methods**
2. Method Name: load
3. Description: Takes object file data as a parameter and returns an **ObjectFile** that represents that data.
4. Parameters: **String** data
5. Requires: A valid “Wi-11 Machine” object file
6. Ensures: the output **ObjectFile** is a valid in-memory representation of the input data
7. Returns: **ObjectFile**
8. Throws: Exception
9. Method Name: validateSymbolName
10. Description: This method accepts a symbol name and determines whether the name is valid.
11. Parameters: **String** name
12. Ensures: If the symbol name is valid, the output **String** will be **null**. If it is invalid, the output **String** will contain an error message for display to the user.
13. Returns: **String**

### ByteOperations

1. Description: A utility class that provides functions to extract and modify binary data.
2. **ByteOperations Methods:**
3. Method: parseHex
4. Description: This method converts a hex number represented in a String to an integer value.
5. Parameters: **String** hex
6. Ensures: hex is a valid hexadecimal number representation.
7. Returns: an integer value representing the input hex value.
8. Throws: Exception
9. Method: extractValue
10. Description: This method extracts bit values from a given integer.
11. Parameters: **int** value, **int** start, **int** end
12. Requires: start < end
13. Returns: an integer value representing the extracted bits.
14. Method: getHex
15. Description: This method gets a string representation of a given integer value.
16. Parameters: **int** value, **int** numCharacters
17. Ensures: All but the least significant hex digits specified by numCharacters are removed.
18. Returns: a string representation of the integer value in hexadecimal form.
19. Method: getBit
20. Description: This method returns true if the bit in “value” at the specified right-aligned index is 1.
21. Parameters: **int** value, **int** index
22. Returns: A **boolean** value indicating if the specified bit is 1.
23. Method: extendSign
24. Description: Extends the sign bit from the given most significant bit.
25. Parameters: **int** value, **int** mostSignificantBit
26. Returns: the sign-extended version of the given input value
27. Requires: 0 < mostSignificantBit < 32
28. Ensures: if the bit indexed by mostSignificantBit is set, all the bits to the left of that bit will also be set.
29. Throws: Exception

### ObjectFile

1. Description: Each instance of this class represents an object file loaded in memory. This class stores an object file's segment name, execution start address, text records, imported and exported symbols, and relocation information.
2. State:
3. private **List<SymbolEntry>** symbolEntries
4. private **List<SymbolEntry>** relocationRecords
5. private **MemoryBank** memory
6. private **SymbolTable** symbols
7. private **boolean** relocatable
8. private **int** startAddress
9. private **String** name
10. **ObjectFile** Methods
11. Method: getSymbolEntries
12. Description: Gets the List of SymbolEntries representing imported Symbols.
13. Returns: **List<SymbolEntry>**
14. Method: getRelocationRecords
15. Description: Gets the **List** of **SymbolEntry**s representing relocation records.
16. Returns: **List<SymbolEntry>**
17. Method: getMemoryBank
18. Description: Gets the **MemoryBank** representing the text records of this **ObjectFile**.
19. Returns: **MemoryBank**
20. Method Name: getSymbols
21. Description: Gets the **SymbolTable** representing the **Symbol**s defined in this **ObjectFile**.
22. Returns: **SymbolTable**
23. Method Name: isRelocatable
24. Description: Returns **true** iff this **ObjectFile** is relocatable.
25. Returns: **true** iff this **ObjectFile** is relocatable.
26. Method Name: getStartAddress
27. Description: Returns the address to start execution at in this **ObjectFile**.
28. Returns: **int**
29. Method Name: getSegmentName
30. Description: Returns the segment name specified in this **ObjectFile**.
31. Returns: **String**
32. Method Name: relocate
33. Description: Relocates this **ObjectFile**'s **SymbolTable**, **MemoryBank**, and start address.
34. Parameters: **int** a, **int** b
35. Returns: **void**

### Error

1. Description: An instance of **Error** describes an error encountered in the assembly of the input program.
2. State: private **int** line, private **String** message, private **boolean** hasLine
3. **Error** Methods:
4. Method Name: hasLineNumber
5. Description: Used to determine whether this is associated with a line number.
6. Returns: true if and only if this error has a line number associated with it
7. Method Name: getLineNumber
8. Description: Gets the line number associated with this **Error.**
9. Returns: 0 if this.hasLine = false; otherwise returns the line number of this **Error**
10. Method Name: getMessage
11. Description: Gets the message describing this **Error** to the user.
12. Returns: the message associated with this **Error**

### SymbolEntry

1. Description: A **SymbolEntry** consists of a symbol, an address, and a bit mask.
2. State:
3. private **String** symbol
4. private **int** address
5. private **int** leastSignificantBit
6. private **int** mostSignificantBit
7. **SymbolEntry Methods**
8. Method Name: getAddress
9. Description: Gets the address of this **SymbolEntry**.
10. Returns: **int**
11. Method Name: getSymbol
12. Description: Gets the name of the **Symbol** associated with this **SymbolEntry**.
13. Returns: **String**
14. Method Name: setAddress
15. Description: Sets the address of this **SymbolEntry**.
16. Parameters: **int** a
17. Returns: **void**
18. Method Name: getLeastSignificantBit
19. Description: Gets the least significant bit to insert the **Symbol** into.
20. Returns: int
21. Method Name: getMostSignificantBit
22. Description: Gets the most significant bit to insert the **Symbol** into.
23. Returns: **int**
24. Method Name: getMask
25. Description: Gets the bitmask for this **SymbolEntry**.
26. Returns: **short**
27. Method Name: getInverseMask
28. Description: Gets the inverted bitmask for this **SymbolEntry**.
29. Returns: **short**

### MemoryBank

1. Description: A **MemoryBank** represents an array of 16-bit memory. It provides functions for reading and writing memory, as well as resolving symbol values and relocating binary data.
2. State:
3. private **Map<Integer, Short>** data
4. The concrete representation of the memory. This dictionary maps 16-bit unsigned addresses to 16-bit signed values.
5. private **int** firstAddress
6. The first address with data in this MemoryBank.
7. private **int** lastAddress
8. The last address with data in this MemoryBank.
9. private **int** startAddress
10. The address (relative to the origin) at which to start execution.
11. **Program** Methods:
12. Method: write
13. Description: Sets the memory cell at the given address to the given value.
14. Parameters: **int** address, **short** value
15. Alters: this.data
16. Requires: 0 <= address < 65,536 and −32,768 < value < 32,767
17. Alters: this.data
18. Ensures: this.data = #this.data + (address, value)
19. Returns: **void**
20. Method: read
21. Description: Gets the value of the memory cell at the given address.
22. Paramaters: **int** address
23. Requires: 0 <= address < 65,536
24. Returns: **short**
25. Method: displayPage
26. Description: Prints the state of the given memory page (bit-shifted all the way to the right) to the given output stream.
27. Parameters: **PrintStream** output, **int** page
28. Requires: output is open
29. Alters: output
30. Ensures: output = #output + [page of memory from this.data]
31. Returns: **void**
32. Method: relocate
33. Description: Relocate the data in this **MemoryBank** from the given starting location to the given new location, using the given relocation records.
34. Parameters: **int** a, **int** b, **List<SymbolEntry>** relocationRecords
35. Returns: **void**
36. Method: resolveSymbols
37. Description: Insert values from the given **SymbolTable**s into the memory locations specified by the given **List** of **SymbolEntry**s.
38. Parameters: **List<SymbolTable>** symbols, **List<SymbolEntry>** symbolEntries
39. Returns: **void**
40. Throws: Exception
41. Method: insertInto
42. Description: Insert the data from this **MemoryBank** into the given **MemoryBank**, overwriting any overlapping data.
43. Parameters: **MemoryBank** bank
44. Returns: **void**
45. Method: getRecords
46. Description: Gets a **String** of text records representing this **MemoryBank**.
47. Returns: **String**
48. Method: adjustBounds
49. Description: Expands the first or last address of the MemoryBank to include a given address.
50. Parameters: **int** address
51. Returns: **void**

### Symbol

1. Description: A **Symbol** is a name-value pairing that has a flag for whether or not the symbol is relocatable or absolute. It also has flags indicating whether the Symbol is imported, exported, or neither.
2. State: private **String** name, private **int** value, private **boolean** isRelocatable, private **boolean** \_isImport, private **boolean** \_isExport
3. **Symbol** Methods:
4. Method: getName
5. Description: Returns the name of this **Symbol**
6. Returns: **String**
7. Method: getValue
8. Description: Gets the value of this **Symbol**
9. Returns: **int**
10. Method: isRelocatable
11. Description: Returns true if this **Symbol** is relocatable, false if not.
12. Returns: **boolean**
13. Method: isImport
14. Description: returns true iff the import flag is set for this **Symbol**, indicating that this **Symbol** must be defined in another linked object file.
15. Returns: **boolean**
16. Method: isExport
17. Description: returns true iff the export flag is set for this **Symbol**, indicating that this **Symbol** will be exported to other linked object files.
18. Returns: **boolean**
19. Method: setExport
20. Description: Sets the export flag for this **Symbol**.
21. Returns: **void**

### SymbolTable

1. Description: An instance of this class maintains a mapping between **Symbols** and their values.
2. State: private **Map**<**String, Symbol**>
3. **SymbolTable** Methods:
4. Method: define
5. Description: Adds a given **Symbol** to this **SymbolTable**.
6. Parameters: **Symbol** symbol
7. Alters: this
8. Ensures: this = #this + symbol
9. Method: define
10. Description: Defines an alias for another **Symbol**.
11. Parameters: **String** alias, **String** target
12. Alters: this
13. Ensures: this = #this + alias
14. Method: hasSymbol
15. Description: Returns true if this table contains a **Symbol** that matches the given name.
16. Parameters: **String** name
17. Returns: **boolean** true if and only if name matches a **Symbol** in this
18. Method: get
19. Description: Gets the **Symbol** mapped to the given name, or null if none exists in this.
20. Parameters: **String** name
21. Returns: **Symbol** corresponding to name input
22. Method: size
23. Description: Gets the number of defined **Symbols** in this table.
24. Returns: **int** number of defined **Symbols**
25. Method: getSymbols
26. Description: Gets the set of all address/value pairs in this table.
27. Returns: **Collection<Symbol>**
28. Method: relocate
29. Description: Relocate all relocatable symbols from a to b.
30. Parameters: **int** a, **int** b
31. Returns: **void**