PROGRAMMER’S GUIDE

Table Of Contents

***Content: Page Number***

Introduction **2**

Data Structures **3**

Module Description **4**

Module Overview **7**

Error Handling **9**

Data Element Dictionary **10**

Introduction

This programmer’s guide is a thorough description of how this linker is implemented. It is provided as a development road-map for the linker and is intended to aid to and provide for quick modification of the program. Descriptions of the different sections are included below:

1. **Data Structures**

-Provides a description of how the assembler is represented

1. **Module Descriptions**

-Provides a detailed description of each module

1. **Modules Overview**

-Describes relationships between modules and shows the overall flow of the program.

1. **Error Handling**

-Describes how errors are handled within the program

1. **Data Element Dictionary**

-Listing of all shared variables in the program; Provides the name, scope, type, declaring module, and purpose of each variable

Implementation of this machine is carried out using RESOLVE/C++ however strict RESOLVE conventions are not followed, for example parameter modes are not specified with function arguments and the word “object” does not precede all object declarations. RESOLVE foundation types are used in place of C++ primitives and RESOLVE components are used as much as possible. Function declarations follow the format *firstSecond()* where the first word/abbreviation is lowercase and the second word/abbreviation, if present, has a capital first letter followed by lower case letters. Object declarations follow the same convention as functions and class names are similar however the first letter of the first word/abbreviation is capital as well. Interfaces are not used therefore it is recommended that new classes/data-structures subclass an existing one. RESOLVE does not provide components that allow for easy storage and referencing of external symbols, therefore, have added a data structure that allow us to more easily access these elements. This data structure allows us to extract, reference, and relocate these elements within the linker.

Data Structures

*symbol\_table.h*

*Symbol\_Table* acts as workable reference table for the linker program. This table is built during pass1() of the linker, and referenced for information in pass2(). Since this structure was originally created for use with the assembler an ‘Other’ field exists, however, in the linker it is not used for storing information and therefore, its contents are ignored.

*Symbol\_Table* does not provide in-depth operations allowing addition and change-of-base of table elements; however, these operations are needed for our purposes and therefore are implemented externally as modules. This class was constructed using the Resolve/C++ components Partial Map and Array, (more specifically, the table is a partial map of arrays.). The partial map was instantiated with *Partial\_Map\_Kernel\_3\_C* because of the improved search algorithm which improved overall search speed. The arrays were instantiated with *Array\_Kernel\_1\_C.*

In the class *Symbol\_Table* , the fields of the partial map, *Symbol\_Template,* of array, Symbol\_Data, can be represented as:

*(Name, (Value (in hex), Other))*

From these fields, the follow operations were created to allow easy access to the data initialized in the class *Symbol\_Table*:

**void addSymbol(Text &, Text &, Text & )**

­- Note: Adds a new symbol to the table

**Text getOther(Text &)**

- Note: Returns ‘Other’ field from table, this field was primarily used for

relocation.

**Text getValue(Text &)**

- Note: Returns Value field from table.

**void removeAny(Text &, Text &, Text & )**

- Note: Returns fields of data with random selection.

**Integer size()**

- Note: Returns size of table (Number of symbols in source program)

**Boolean check(Text &)**

- Note: Returns true if t is in table.

Note: Brief descriptions of operations are results of direct similarity to resolve operations.

Module Description

Boolean checkFiles(Command\_Line\_Handler cmdLine, CharacterO\_Stream stdOut)

The *checkFiles()* operation checks whether the filename from the input line exists. If the file does exist, the user is asked whether or not he would like to overwrite the file. If the file exists, then several checks are performed. If he chooses ‘y’ or ‘Y’, then he decides he would like to overwrite the file, and false is returned from the operation. If he chooses anything else, ‘n’ or ‘N’ in particular (for no), then another check is performed. The file is then checked to see if it is writeable. If it is not, then an error is reported and false is returned.

If the file does not exist, then an error is reported and false is returned. If the file does exist, one more check is performed and that is to see if the file is readable. If it is not, then an error is reported and false is returned. All of these checks are done for the two output files from the command line. If both of these files pass all the checks and errors are not found or reported, then at the end of the operation true is returned as a Boolean.

Boolean representsAnInteger(preserves Text& value)

The operation *representsAnInteger()* was created to allow a simple check of record syntax. The operation first checks for a length greater than zero, if so it checks the first character to determine if the vale represented is negative, indicated by ‘-‘. Then if loops through all the to determine whether it contains any non integer ascii characters. If and only if the parameter contains an integer value, the operation will return true.

Boolean representsHex(preserves Text& hex)

The operation *representsHex()* was created to allow a simple check of record syntax. The operation first checks for a length greater than zero, if so it then loops through all the characters to ensure that they are [0-9] | [A-F] | [a-f] based on the ascii character encoding scheme. If and only if the parameter contains a positive hexadecimal value, the operation will return true.

Integer toInteger(preserves Text& value)

The operation *toInteger()* reads in a text operand and converts the value to an integer. It is important when using *toInteger()* that the programmer has already run *representsAnInteger()* to ensure that the parameter provided represents an integer. The operation will loop through each element of the given text operand and return the appropriate integer value.

Text toHexAddress(Integer i)

The operation *toHexAddress()* is used to convert a decimal integer with the range

0<= i <= 255. To do this a for-loop that will run through twice is implemented. In each pass will take mod16 of the number and then add the hex value to a temporary Text object. After the first pass the number will be divided and reassigned to itself. After the operation finishes the for-loop it will return the temporary Text object.

Integer toDecimal(preserves Text& hex)

The operation *toDecimal()* is used to convert a hexadecimal value to a decimal integer. It is required that the incoming Text object contain legal hexadecimal characters, therefore, it is recommended that the programmer call *representsHex()* before calling this operation otherwise unpredictable behavior will result. The computed integer equivalent of the hexadecimal parameter will be returned.

Text getSymbol(preserves Text& t)

The purpose of the *getSymbol()* operation is to obtain the symbol from an ‘E’ record from the object input file. The symbol of the line is the first six non-space characters on the line after the ‘E’. The first six non-space characters are added to a temporary Text object. Once the first space character is encountered, the operation is broken and the temporary Text object is then returned by the operation.

Text getSymbolValue(preserves Text& t)

The purpose of the *getSymbolValue()* operation is to obtain the symbol value from an ‘E’ record from the object input file. The symbol value is the two hex characters on the line after the ‘E’ and the six characters (spaces or not) which comprise the symbol. These two characters are added to a temporary Text object which is then returned by the operation.

Text getSegmentName(preserves Text& t)

The purpose of the *getSegmentName()* operation is to obtain the segment name from the header record from the object input file. The segment name is the first six non-space characters on the line after the ‘H’ and the execution-begin-address. The first six non-space characters are passed through and added to a temporary Text object. Once the first space character is encountered, the operation is broken and the temporary Text object is then returned by the operation.

Text getSegmentLength(preserves Text& t)

The purpose of the *getSegmentLength()* operation is to obtain the segment length value from the header record from the object input file. The segment length is the last two hex characters on the line before the ‘M’. These two characters are added to a temporary Text object which is then returned by the operation.

Module Overview

Linker Execution

Execution of the linker begins in the *main()* function of linker.cpp. Main controls the checking of the arguments to ensure that they are files and have proper permissions set. From here output to the *error-log* and *executable-output* files are opened. Next the module *linker()* is called. The behavior of *linker()* and the modules called within it are described below. After *linker()* has returned, output to the *error-log* and *executable-output* files are closed and the program terminates gracefully.

void linker()

The module *linker()* controls calls to the various main parts of execution; modules *pass0()*, *pass1()* and *pass2()*. If either *pass0()* or *pass1()* return false then this module will terminate without calling the remaining passes. Termination of a pass means that a fatal error has occurred and information about the error can be found in the *error-log*.

Boolean pass0()

The job of *pass0()* is to compute the memory footprint size (global variable *footprint*) and get the Initial Program Load Address (global variable *IPLA*) from the user. In addition it also checks to make sure that all of the object files are relocateable and that one and only one object file specifies “Main”. The module begins by looping through each object file, reading in the header, checking if the segment name is “Main” and adding the segment length to *footprint*. If in this loop it finds more than one segment named “Main” it will report an error to the *error-log* file then return false, halting execution of the linker. After finishing the loop it will check to make sure that a “Main” segment was found; again, if not, an error will be reported to the *error-log* file and then return false. Next, *footprint* is checked to make sure that it does not exceed the space available in memory (256 words). If this check fails an error will be reported to the *error-log* file and then return false, however, if this check passes then the user will be asked to input a hexadecimal value for the *IPLA*. If the input value does not represent a hexadecimal value or if it is out of range (input < 0 | 255 < input) then the user will continue to be asked until an appropriate value is given. If the input value plus *footprint* is greater than 255 (0xFF) then wrap around will occur and the linker will continue assigning memory locations at 0 (0x00) after location 255 (0xFF) has been filled.

Boolean pass1()

Entry symbols, the program name and symbols defined in ‘E’ records, must be added to the External Symbol Table (global variable *extSymbolTable*) ; this is the job of *pass1()*. This module loops through all object files adding the program name with the current Program Load Address (PLA) to the *extSymbolTable* and looks for records beginning with ‘E’ to add the associated symbol and value to the *extSymbolTable*. These ‘E’ records will occur immediately after the header record and before any text records. The module checks the program name to see if it is “Main”, if so then the address where execution is to begin is set (global variable *executionBegin*) by adding the PLA to the execution begin address from the header record. Next the module adds add symbols defined with ‘E’ records to the *extSymbolTable*. The symbol on the ‘E’ record line will be parsed using *getSymbol()* and the value associated with it will be parsed using *getSymbolValue()*. The symbol will be checked to see if it already defined in the *extSymbolTable,* if it is already defined then an error will be reported to the *error-log* file and then the module will return false. If it is not defined then the symbol and value added to the current PLA will be added to the *extSymbolTable*. Once the first text record is encountered the loop will be finished with the current file, the PLA will be updated for the next program by adding the current segment length to it.

void pass2()

Similarly to the operation of *pass1()*, *pass2()* will check for external symbols occurring at the end of text records. These symbols must be removed and the associated value must be added to the ‘S’ field on the respective text record line. In addition to this the memory address of each text record is updated to reflect its proper location in memory relative to the *IPLA* and the text record is output to the *executable-output* file. This module begins by outputting the header record to the *executable-output* file. Information contained in the header is gathered from *executionBegin*, *IPLA* and *footprint*. Next, the module begins looping through each file and updates the address of each text record by adding the current PLA to the address. If the length of the text record is greater than eight, then the record contains either a ‘M’ or a ‘X’ and the ‘S’ field must be updated; otherwise, the text record is complete and it is output to the *executable-output* file. If the 9th character (index 8) of the text record is a ‘M’ then the ‘M’ is removed, the current PLA is added to the ‘S’ field and the record is output to the *executable-output* file. However, if the 9th character is a ‘X’ then the ‘X’ is removed and the symbol following the ‘X’ is removed and stored in a temporary variable. If this symbol is not defined in *extSymbolTable* then an error is reported to the *error-log* file and then the module will return false. Otherwise, if the symbol is defined then the value of that symbol plus the current PLA will be added to the ‘S’ field and the record will be output to the *executable-output* file. After updating each text record in the object file the PLA is updated for the next program by adding the current segment length to it.

Error Handling

Group I: Input from Command Line

There are several errors checked from the input command line, all of which are checked with ‘IF’ statements. The first error which is checked for is if less than three arguments are used, being the *executable-output*, *error-log* and at least one *object-file*. If the number of arguments is less than three then an error is output to the terminal. This error is also fatal and ends the program abruptly.

Next, the program checks to see if the output files are writeable. If they are not, an error is output to the terminal, as well as to the listing file. This error is also fatal and ends the program. The next error, which is also fatal, is to check if the filenames from the command line (input 2, 3 and 4) are directories or not. If the filenames are in fact directories and not a file, then an error is output to the terminal and the program is ended.

If the filenames from the command line are determined to not be directories, then they are run through the *checkFiles()* operation. This operation checks for several errors within itself. The first error it checks for is if both the file exists, and the file is not writable, then an error is output to the listing file. The operation is then exited from by returning a false Boolean and an error is returned to the terminal. This error is also fatal. The next two errors which are checked for are if the files do not already exist. ‘If’ statements then check to see if either the file does not exists, or if it does exist and is not writeable. If either of these checks prove true (the file does not exist or the file is not writeable), then an error is reported to the listing file and a false Boolean is returned from the operation. Once exited, an error is output to the terminal window and the program is exited as these errors also are fatal.

Group II: Pass Zero, Pass One, Pass 2

Errors contained in *pass0()*, *pass1()* and *pass2()* are all fatal and halt execution of the linker upon encounter. The cause of these errors has been explained above in the respective module explanations. For more information on explicit error messages please refer to the User’s Guide.

Data Element Dictionary

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Variable Name | Local/Global | Type | Declaring Module | Purpose/Use |
| cmdLine | Global | Command\_Line\_Handler | linker.cpp | Contains the files passed as arguments to the program |
| stdOut | Global | Characer\_Ostream | linker.cpp | Used to display errors and information to the terminal window |
| execOut | Global | Characer\_Ostream | linker.cpp | Object code output; input for the simulator |
| errLog | Global | Characer\_Ostream | linker.cpp | Output to a file where errors are reported |
| IPLA | Global | Integer | linker.cpp | Initial Program Load Address; declared by user |
| footprint | Global | Integer | linker.cpp | Total number of words in memory that the program will occupy |
| executionBegin | Global | Text | linker.cpp | The address where execution is to begin |
| extSymbolTable | Global | Symbol\_Table | linker.cpp | External Symbol Table (see Data Structures) |