PROGRAMMER’S GUIDE

Table Of Contents

***Content: Page Number***

Introduction **2**

Data Structures **4**

Module Description **8**

Module Overview **12**

Error Handling **18**

Data Element Dictionary **22**

Introduction

This programmer’s guide is a thorough description of how W10-560 Machine assembler works. It is provided for the knowledgeable user who finds some need to modify the program quickly and without direct contact from the original software designer. 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 among classes and shows the overall flow of the program.

1. **Error Handling**

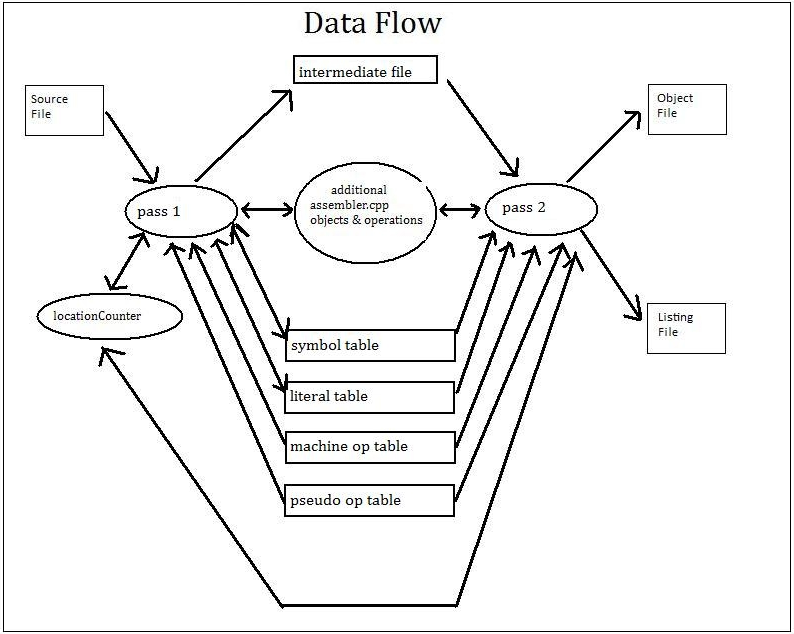
-Describes how errors are handled within the program

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 use 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 have components that allow for easy storage and referencing of machine operations, pseudo operations, symbols, and literals. We have added data structures that allow us to more easily access these elements. These data structures allow us to extract, reference, and relocate these elements within the assembler.

Below is a simple diagram description of our program data flow. This diagram should not be used for explicit definitions of objects, classes or methods, but rather should be used to gain a basic understanding of processes before progressing through this guide.



Data Structures

*machine\_table.h* and *pseudo\_table.h*

The classes *Machine\_Table and Pseudo\_Op\_Table* act as the reference tables for the assembler program. The **purpose** of these tables is to provide organized informational “libraries” capable of simple checking and data retrieval. These two tables were created with identical structure and only differ in set size and operations provided.

Both of these classes were constructed using the Resolve/C++ components Partial Map and Array, (more specifically, the tables are partial maps of arrays.). The partial maps were instantiated with *Partial\_Map\_Kernel\_3\_C* because of the improved search algorithm which improved overall integration speed with assembler.cpp. The arrays were instantiated with *Array\_Kernel\_1\_C.* By choosing the built in components from the Resolve catalog, our table’s operation algorithms were greatly simplified because of parallel operations from the components themselves. (For example, the class instruction Boolean check() is simply an extension of partial map operation isDefined().)

In the class *Pseudo\_Op\_Table* ,the fields of the partial map, *Pseudo\_Data,* of array, Array\_Of\_Pseudo\_Data, can be represented as:

*(Operation, (Length, Format))*

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

**Integer getLength(Text &)**

-Note: Returns given length integer for the size of the desired operation.

**Text getFormat(Text &)**

* Note: Returns addition information for operation.(Not used in program, but kept for range if adapted)

**Boolean check(Text &)**

- Note: Used to check table existence for operations

In the class *Machine\_Table,* the fields of the partial map, *Table\_Data,* ofarray, Array\_Of\_Data, can be represented as:

*(Operation, (Opcode, Length, Format))*

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

**Character getOpcode(Text &)**

- Note: Returns one character machine opcode

**Integer getLength(Text &)**

- Note: Returns length of the opcode

**Text getFormat(Text &)**

- Note: Returns five character machine code formatted with opcode.

**Boolean check(Text &)**

- Note: Checks if opcode is in *Machine\_Table*

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

*literal\_table.h* and *symbol\_table.h*

Like the classes *Machine\_Table* and *Pseudo\_Op\_Table, Literal\_Table* and *Symbol\_Table* act as workable reference tables for the assembler program. However, these tables are built during pass1() of the assembler, and referenced for information in pass2() unlike the previous “pre-made” designs. The **purpose** of these two tables is to provide accessible data storage systems for the assembler to access in assembler.cpp. Specifically, *Literal\_Table* is built to handle designated literals, where as *Symbol\_Table* is built to handle assembler language symbols. The class operations following reflect specific needs of information.  *Literal\_Table* and *Symbol\_Table* have limitations of a fifty Literal max and a one-hundred symbol max respectively, however, the appropriate error checking was done in *assembler.cpp* because of the need to change program error locations and adapt reactionary measures.

*Literal\_Table* and *Symbol\_Table* both required more in-depth operations allowing addition and subtraction of table elements. Like the previous tables, both of these classes were constructed using the Resolve/C++ components Partial Map and Array, (more specifically, the tables are partial maps of arrays.). The partial maps were 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 *Literal\_Table* , the fields of the partial map, *Literal\_Template,* of array, Literal\_Data, can be represented as:

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

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

**void addLiteral(Text &, Text &, Text & )**

- Note: Allows all values to be added to table at once

**Text getAddress(Text &)**

- Note: Returns address in Hex

**Text getValue(Text &)**

- Note: returns the value in Hex

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

- Note: Remove any is used instead of remove because addressing is

independent of table order

**Integer size()**

- Note: returns size of the table

**Boolean check(Text &)**

- Note: checks if literal is defined in table

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 following 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.

*Intermediate\_Table.H*

The **purpose** of the *Intermediate\_Table* class is to act as a temporary bridge between pass 1 and pass 2 in the *assembler.cpp* file. This table accepts input copies of each line that is read in from the original input file, *assemblySource.* allowing extraction of these files in-order in pass 2. Since this process of inserting and extracting mirrors the FIFO property of queues, a RESOLVE/C++ *Queue\_Of\_Text* is used to construct the table. *Intermediate\_Table* is instantiated with *Queue\_Kernel\_1a\_C.* The *Intermediate\_Table* class has 3 public functions and one constructor *Intermediate\_Table()*. This is left empty because when the *Intermediate\_Table* class is created, it will automatically begin as an empty queue. The other three functions and their uses are listed below.

**void add\_inter\_line(Text &t)**

-Note: Will add a new line to the *Intermediate­\_Table* object.

**Text get\_inter\_line()**

-Note: Will produce a line of text in the order that the original lines were placed into the *Intermediate\_Table* object.

**Integer inter\_size()**

-Note: Will give the size of the *Intermediate\_Table* object.

Module Description

Text getLabel(preserves Text& t)

The purpose of the *getLabel()* operation is to obtain the label from an input line from the assembly source file. The label of the line is the first six characters on the line. If the first character is a space, then the operation is broken and exits. If it is not a space, then the six characters are passed through and added to a temporary text which is then returned by the operation. Before the characters are added, they are checked to see if their ascii value is a readable value. If any of them are not, then an error is reported.

Text getOperation(preserves Text& t)

The purpose of the *getOperation()* operation is to obtain the operation from the input from the assembly source file. The line is first checked to see if there are enough characters in the line in order to have an operation. If there are not enough characters, then an error is reported. If there are, then characters 8 and 9 are taken out, and if character 10 is not a space then it is taken as well. Those are all put together in a text object and returned from the operation.

Text getOperands(preserves Text& t)

The purpose of the *getOperands()* operation is to obtain the operands from the input from the assembly source file. The get operands operation is very simple. It simply runs a loop checking characters after position 13 in the input line until either a space is found or the end of the line is reached. The characters are read and put into a text object, and that object contains the operands and is then returned from the operation.

Text getNextOperand(alters Text& t)

The *getNextOperand*() operation is used to progress through an operation’s operands. The operation will remove characters from the operand and add them to a temporary Text object. Until it reaches the end of the operand or it reaches a “,”, “(“, or “)”. If the operation stops because it reaches one of the symbols, it will return the temporary text object. If it stops because it is at the end of the line it will return nothing.

Boolean representsAnInteger(preserves Text& t)

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 it 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.

Text getCCDOperands(preserves Text& t)

The operation *getCCDOperands()* was created in addition to *getOperands()*because of the unique requirements that CCD provides. Specifically, *getCCDOperands()* removes the two characters that the CCD requires in record positions 14 and 15. It also provides an error check to account for a potential missing operand character.

Integer toInteger(preserves Text& t)

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.

Text toHexValue(Integer i)

The operation *toHexValue()* to convert an integer with a range of -219 <= i <= 219-1 to a 5 character hexadecimal number. The operation works similarly to *toHexAddress()*, except the for loop runs through 5 times instead of twice. If the number is negative it will take the 2’s complement of the number by using another for-loop. The operation will then return the temporary Text object that was produced in the for-loops.

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.

void putListingFileHeader()

This section will cover the *putListingFileHeader()* procedure and how and what is put to the listing file. The *putListingFileHeader()* is an operation in *assembler.cpp*. This outputs the header of the *listingFile* by using 8 text objects of *header0-header7*. Below is pseudo code of how the 8 text objects will look:

*Text header0 = "+-------------------------------------------------------------------------+\n";*

*Text header1 = "| Listing |\n";*

*Text header2 = "|--------------------------------------------------------------------------|\n";*

*Text header3 = "| Object Code | Source |\n";*

*Text header4 = "|------------------ +-----------------------------------------------------|\n";*

*Text header5 = "| Loc Data Rel | Rec Label OP Operands/Comments |\n";*

*Text header6 = "|(hex) (hex) |(dec) |\n";*

*Text header7 = "|-----------------+------------------------------------------------------+\n";*

These text objects will then be output to the listing file in order to create the heading.

The contents of *listingFile* will be either the data to fill in the columns of the header as shown above or error messages. *Pass2* is the only place in *assembler.cpp* that will output the data needed to fill in the listing table. The other procedures (*pass1*, *pass2,* *getLabel*, *getOperation*) will all produce error messages. These error messages will be covered later in the error handling section. This section will only discuss how the correct inputs are put to *listingFile*.

The outputting to *listingFile* that occurs in *Pass2* will occur after the machine code line is assembled. How output is written to *listingFile* will depend on whether the operation in the current line is a machine op or a pseudo op. If the operation is a machine op every line will be put to *listingFile* the same way. *Pass2* will use the original line that was read in from *intermediate\_table* to produce the columns under *Source* and a copy of the assembled line (*lineTemp*) to produce the columns under *Object Code*. *Pass2* will first determine how many spaces to put between the *Rec* (number of records) and *Label* columns. Once it determines the number of spaces to put, it will remove the ”T” at the beginning of *lineTemp* and then put three spaces between the address and the rest of the line. Then starting at the position 10 of the *lineTemp* *Pass2* will add spaces until it hits the center divider which is at position 14. Then two spaces will be added at position 12 of the original line to move the operation under the *OP* column. Finally the lines will be output to *listingFile* using the code:

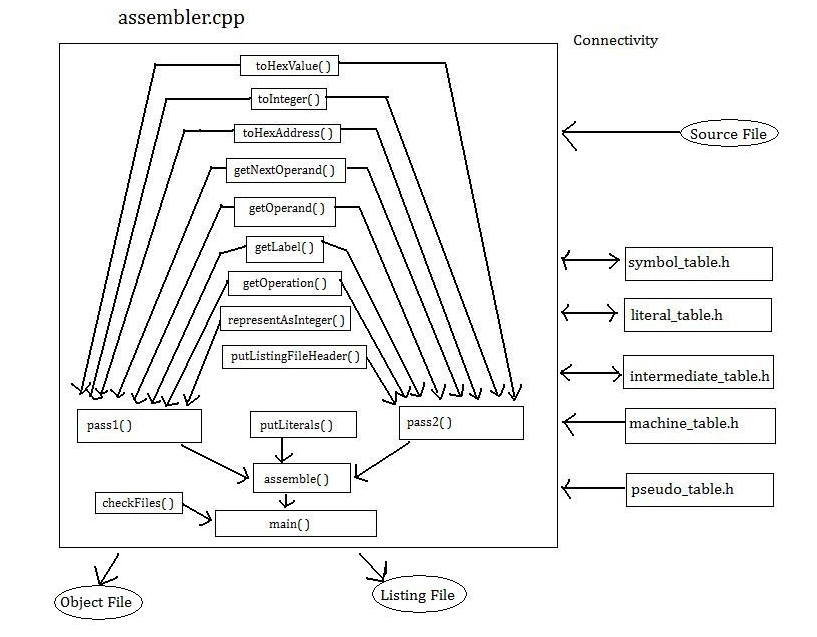
*listingFile <<"| " <<lineTemp <<" | " <<numRecords <<spaces <<line <<'\n';*

If the operation in the line is a pseudo op how *listingFile* is written to will depend on which pseudo op is being called. If the operation is “NMD”, “CCD”, “RET”, “GTC”, or ”PTC” the process of outputting to *listingFile* will be exactly the same as described above with the machine code ops. For “RES”, “ORI”,”END”, and “EQU” two lines will be added to position 12 of the original line to move the op code then output will be written to *listingFile* in the following format:

*listingFile <<"| | " <<line <<'\n';*

Module Overview

Below is the connectivity diagram of *assembler.cpp*. This shows how all the data structures and modules connect and relate with each other.



Assembler Execution

The execution of the *assembler.cpp* file begins with opening the necessary input and output files. The *assemblySource* input file and the *objectOut* and *listingFile* output files are open using the *Open\_External()* operation. The operation *assemble()* is then called. The *assemble()* operation first outputs to the *stdOut* to let the user know that the program is assembling. Three more operation are then called in the order of *pass1()*, *pass2()*, and *putLiterals()*, with *pass2()* and *putLiterals()* only being called if *pass1Error* returns false. These three operations will be discussed below. After *assemble()* is done running the assembler will close the *assemblySource*, *objectOut*, and *listingFile* files.

void pass1 ()

Pass one was created because of potential branching difficulties created from the allowance of symbol back referencing. Pass one will read through the input source file, store all symbols and their related values into a symbol table, relocate the literal values and calculate locations, and lengths of certain operations. With this information, Pass two is capable referencing any existing symbol, (given assembler language restrictions) and assembling the program. In addition to its direct responsibilities listed above, Pass one will also check for errors in the input file. The processes of Pass one are described in further detail below.

The first thing that pass one accomplishes is reading through lines of code from the assembly source file until the first non comment line of code is found. The operation of that line is checked and expected to be an ‘ORI’ pseudo operation. If the operation is anything but an ‘ORI’ operation, then an error is reported. After the ‘ORI’ pseudo operation is found as the first non commented line of code, pass1 () will check for any ENT or EXT pseudo operation, which must precede all other machine or pseudo operation(excluding ORI).It will then store each ENT operand symbol as appropriate. Each ENT operand is stored with an ‘&’ concatenated to the front of the symbol name to maintain the same processes for symbol handling in the original assembler code main. Additionally, the symbol will be stored with an ‘E’ in the “other” field to describe the symbols relocatability and ENT status. Next, a loop is used to parse every line of code until an ‘END’ pseudo operation is found. If the last line of code is found to be any operation except for an ‘END’ pseudo op then an error is reported. The operations are extracted by using a separate method called *getOperation(Text& T)*.

Within the while loop (condition to exit is ‘END’ pseudo op), lines of code are extracted one by one from the input file. The lines are then first checked to see if they are a comment line or empty line. If they are a comment line, which means they start with a semicolon, they are ignored and the next line of code is taken out. The comment lines are ignored because comment lines are used to pass a message, or to communicate with another person who will look at the code, and the program does not need to try to interpret them. When a non-comment or empty line is found, several things are done. One of those things which are done is to check the operation field and see if the location counter should be advanced. If the op field is a machine operation, then the location counter is increased by one, and if the op field is a pseudo operation, it is checked and the location counter is increased appropriately (if at all). If the pseudo operation that is found is an ENT or EXT operation, an error will be reported. While checking the operation field, if it is not in either of the pseudo op or machine op tables, then an error is reported because of an unknown operation.

In the same loop, another task which is accomplished is to create and fill in a symbol table. This is done for every non-comment and non-empty line. When the line is read in from the assembly source file, and it is determined to be a non-comment or empty line and it does not contain an ‘END’ pseudo operation, then it is checked for a symbol using the *getLabel()* operation. If the line were to contain a symbol it would be in the beginning of the line, in the first six characters of the line. If the line contained a symbol, then it is extracted with the *getLabel()* operation. Once the label is taken out, the symbol table is checked to see if there is already a label in it which is the same. If there is a similar label already in the table, then the symbol is checked as to whether it is an ENT symbol stored previously. If it was stored previously the symbol will be updated with the given location of usage. If it is not an ENT symbol an error is reported. If the label is not already in the symbol table then it is added as a pair along with the location counter as its position.

The third task which is accomplished is the lines which can be assembled are checked for literals, and if there are literals they are added to a literal table. The literals are found by using the *checkOperands()* operation to first get the operands, then by checking the first character of the literal to see if it is equal to ‘=’. If it is, then it is a literal. Then the literal is checked against the literal table. If it is in the literal table then nothing is done. If it is not in the literal table, then it is added with its literal value along with its defining text.

At the end of the loop, the next line is taken until it is not a comment, empty line or abrupt end. As explained before, comment lines start with a ‘;’. Empty lines do not have any size and are not processed either. When a non comment or non empty line is extracted from the assembly source file, the loop is repeated as explained above for that line. If the unexpected circumstance happens where the file ends abruptly and there are no more lines of input (before an ‘END’ pseudo operation is found)

void pass 2

The two responsibilities of Pass 2 are to write to the two output files, *objectOut,* which contains the machine code, and *listingFile*, which contains the listing table

.

The first thing Pass 2 does is produce the *listingFile* bycalling the *putListingHeader()* operations.Producing the header record is the next process Pass 2 does. The line

*objectOut <<"H" <<executionBegin <<segmentName <<loadAddress <<length;*

writes to the *objectOut* output file using the text objects *executionBegin*, *segmentName*, *loadAddress*, and *length*, the last three of which were produced in Pass 1. Pass 2 will now begin now run in a while loop until all the text records are output to the *objectOut* file

The first record outputs will be “ENT” related if an ENT exists. Pass 2 will read in from the *interTable* using the *get\_inter\_line* operation and create ENT records for any ENT symbols in the assembler source program. The checking process involves concatenating an ‘&’ to the operands, checking the operands in symbol table and then removing the desired symbols and their addresses. With this information the ENT records are produced as shown:

*assembledLine.Add(0,'E');*

*assembledLine.Add(“1-6”,”Symbol Name”);* (Process Summary, not actual coding)

*assembledLine.Add(j,loc\_val[0]);*

*assembledLine.Add(j+1,loc\_val[1]);*

Pass 2 will then read in more lines from the *interTable* using the *get\_inter\_line* operation. After the line is read in the operation will be extracted from the temporary line and put into the text object *op*. The program now begins assembling the text records. This process starts by obtaining the address (in hex) of the current line. This is then placed in the text object *assembledLine* by using the following code:

*assembledLine.Add(0,'T');*

*assembledLine.Add(1,address[0]);*

*assembledLine.Add(2,address[1]);*

This adds the letter “T” and the address to the first 3 positions of *assembledLine*. The next step is to check what kind of operation is in the *op* object. The two possibilities for what *op* can be, either a machine op or a pseudo op, divided by an if-else statement.

Note: In both Machine and Pseudo Operation sections below symbols will be processed while assembling the ‘T’ records. During this ‘T’ record assembly, a symbol checking algorithm is written to find all EXT symbols. This algorithm checks every symbol in the symbol table for an ‘X’ in the “other” field. If this ‘X’ is found, the appropriate ‘T’ record will be appended as follows:

*assembledLine.Add(9,'X');*

*assembledLine.Add(“10-15”,”Symbol Name”);* (Process Summary, not actual coding)

` **MACHINE OPS**

If the operation is a machine op, pass2() will then use the *getOp()* operation to retrieve the machine op code from the *Machine\_Table* object *machineTable*. This machine op will then be added to position 4 in *assembledLine*. Pass 2 then retrieves the first operand using the *getNextOperand()* operation to retrieve the R field and checks to see if the R field is a symbol. If so, the symbol is retrieved from *symbolTable* using the *getValue()* operation. Then, whether the operand was a symbol or not, it will be added to position 5 of *assembledLine*. Pass 2 then proceeds to initialize the X field by placing a “0” in position 6 of *assembledLine* so it can move on to assemble the S field first.

The first process in assembling the S field is to get is to use *getNextOperand()* to retrieve the S field from the object *operands*. This operand is then checked to see whether it is a symbol, a literal, or a normal S field entry. If the operand is a literal then Pass 2 will use the *getAddress()* operation to retrieve the address from the *literalTable*. The two character value is then placed into positions 6 and 7 of *assembledLine*. If the line is relocatable then an “M” will be added to position 8 of *assembledLine*. If the operand is a symbol then the process for a literal will be repeated except the operation *getValue()* will be used to remove the symbol from *symbolTable*. If the operand is a normal S entry then Pass 2 will check to make sure it is within the allowed memory range (0 <= Memory Location <= 255) and add it to positions 6 and 7 of *assembledLine* is with this range. The next process in Pass 2 is to assemble the X field.

The process of assembling X again begins with using *getNextOperand()* to retrieve the X field from *operands*. Pass 2 first checks to see if the X field is an operand. If it is a symbol, the value of the symbol will be retrieved from *symbolTable* and will replace the “0” that is currently in position 5 of *assembledLine*. The operand is then checked to see if it is a 1, 2, or 3. If the operand is one of those, position 4 will be replaced with a 4, 8, or C, respectively. If it doesn’t meet any of the above conditions then the X field will be left alone in *assembledLine*. *assembledLine* is now finally outputted to *objectOut* and the *locationCounter* and *numRecords* are incremented. The last thing that is done is to output to *listingFile* using the process described previously in this document.

**PSEUDO OPS**

The process to assemble a pseudo op line is quite a bit different from the process to assemble a machine op line. The first thing that is done is to check what pseudo op the operation is. This can either be “NMD”, “CCD”, “RES”, “RET”, “GTC”, “PTC”, “ORI”, “EQU”, “END”, “EXT”, or “ENT”. If the operation is “NMD” or “CCD”, the line will be assembled in the same if statement, as these two pseudo ops are closely related. The first step is to retrieve the next operand using *getNextOperand()*. If the operation is “NMD” then the hex value of the integer in the operand will be taken and added to positions 3-7 of *assembledLine*. If the operation is “CCD” the hex value of the two characters will be taken and placed into two temporary text objects, each with two characters in them. These will then be added to positions 3-6 of *assembledLine* with position 7 being assigned a “0”. After the line for the “CCD” or “NMD” operation is assembled Pass 2 will output to *objectOut*, increment the *locationCoutner* and *numRecord*, and then output to the *listingFile* as described earlier in this document.

If the operation is “RES” then the there will be no output to the *objectOut* file but the *locationCoutner* will be affected. The operand must first be retrieved from the *operands* object and checked to see if it is a symbol. If it is a symbol, the value of the operand is retrieved from the symbol table then converted integer. If not a symbol then the operand will just be converted to an integer. If the value of this integer is between the allowed memory range (0 <= memory location <= 255) then the *locationCounter* is incremented by the value of the integer. The *listingFile* is then updated according to the process described earlier in this document.

If the operation is “RET”, “GTC”, or “PTC”, then assembling process will proceed as described below. Since each of these directly relates to a machine op, the only field that needs to be retrieved from the operand is the R field. This is the first process that is done regarding these pseudo ops. Then the Op field and X field in are placed into positions 2 and 3 of *assembledLine*. “RET” stands for return from subroutine, which is the BR instruction with R=3, so Op=C. “GTC” stands for get character, which is the IO instruction with R=1, so Op=B. “PTC” stands for put character, which is the IO instruction with R=3, so Op=B. The X field is then set by using the value retrieved from the operand. The value must be 0,1,2, or 3 and set the X field to 0,4,8, or C, respectively. The S field is now set by making positions 6 and 7 of *assembledLine* equal to “0” because these instructions are not affected by the S field. Pass 2 will output to *objectOut*, increment the *locationCoutner* and *numRecord*, and then output to the *listingFile* as described earlier in this document. If the operations were “ORI”, “EQU” or “END”, no output will be written to *objectOut*, but the *listingFile* will be updated according to the process described early in this document.

The pseudo operations ENT and EXT are both handled separately from the other pseudo operations and are only mention here to give the complete range of pseudo operations. To reiterate, if an ENT is read, it will be handled as shown above. If an EXT is read, it will be “skipped” because it has already been processed in Pass 1().

void putLiterals()

The purpose of the *putLiterals()* operation is to store the literals in the memory locations after the end of the program. It does this by emptying the literal table using a while loop. In the while loop the, *putLiterals()* uses the *removeAny()* function to take any literal out of the table. It then assembles the line by adding a “T” to the front of the line, then the address of the literal in hex, then the value of the literal in a 5 character hex number. At the end of every iteration the new line is output to the *objectOut* file.

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 whether or not four arguments are used, one being the name of the program (assembler), and the next three being the assembly source, object output and listing output respectively. If the number of arguments is not four 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: Input Records from the Assembly Source File in Pass One

The next group of errors which are checked for are the syntax of the input from each line of the assembly source file. These checks are all completed in Pass 1 of the assembler. The first job done by pass one of the assembler is to read in the lines until the first non comment line is found. Pass one reads in lines one by one, and checks them to see if they are comments (which means they begin with ‘;’). If there are no non comment lines then an error is reported to the listing file.

Once the first non comment line is read in, the operation code is checked. If it is not ‘ORI’, an error is reported to the listing file and the program returns from pass one. If the op code is ‘ORI’, then an ‘if-else’ statement checks to see if there is a label or not. If there is not a label, then an error is output to the listing file. However, this error is not fatal. If there is an operand, then it is converted to an integer. If the integer is less than zero then an error is reported to the listing file because the load address needs to be a positive location. This error is fatal and pass one is exited.

The next bunch of errors is checked for in the main pass loop of the first pass of the assembler. The first error which is checked is to see if the record count is greater than or equal to the maximum number of records. If this is true, this error reports a line to the listing file and is also fatal. The operation code of this record also needs to be either a pseudo operation or a machine operation. An ‘If’ statement checks to see if this requirement is met, and if it is not then an error is output to the listing file and pass one is returned from since this error is also fatal. When the symbol is read in during pass one, an error is reported if there is a symbol which is already defined in the symbol table. This error is reported by outputting to the listing file and exiting from pass one.

Next specific checks are done depending on the operation code. If the operation code is ‘EQU’ then there must be both an operand as well as a label. If the record does not contain either of these then an error is reported to the listing file. Both of these errors are fatal and exit from pass one immediately. If the ‘EQU’ record does contain both a label and an operand then a check is done against the operand. If the operand does not contain either a symbol or an integer, then an error is output to the listing file and pass one is exited immediately as this error is fatal. Once again, all of these errors are checked with ‘If-else’ statements.

Another ‘If’ statement checks to see if the operation is ‘RES’. If it is, then the only error we checked for was if the operand field contains an integer or not. If the operand was in fact an integer, then no error was reported but if the operand field did not contain an integer then an error was reported to the listing file. This error is also fatal.

Another check is done to see if the operation is in either the pseudo operation table or the machine operation table. These tables contain the only known operation which the machine can perform or interpret. An ‘If’ statement checks to see if the current record’s operation is in either of these table, and if it is not an error is reported to the listing file. This error does not immediately exit pass one because it is not fatal.

All of these error checks are done for every line until the last record is extracted from the assembly source file. When the last record is reached, the operation must be ‘END’. If the operation is not ‘END’, then an error is printed to the listing file and pass one is exited. This error is fatal. If the operand is in fact an ‘END’ operand, then several more error checks are performed on it. A check is done to see if the operand on the ‘END’ record is relative or not. If the operand is relative and the program is relocatable, then nothing is done because that is the correct format. If the program is relocatable and the operand does not agree, then an error is printed to the listing file. This error is not fatal and does not cause an immediate exit.

The next set of errors is for checking the size of appropriate tables. The three checks here test the number of records, the size of the literal table and the size of the symbol table. If the size of the literal table is greater than or equal to the maximum number of literals allowed; then an error is reported to the listing file. If the size of the symbol table is greater than the maximum number of symbols allowed; then an error is also printed to the listing file. If the record count is more than or equal to the maximum number of records then yet another error is printed to the listing file. Although all of these three errors report to the listing file, none of them are fatal to the program execution.

The *getLabel()* and *getOperation()* which are used by pass one also report errors when appropriate. The *getLabel()* operation checks for two errors. While the operation checks in a for-loop to see if the characters are characters which are allowed in the label, if any of the characters are incorrect an error is reported. The way that the error is reported is it is printed to the listing file, and an empty string is returned as the label. If all of the characters are correct, one more check is done before the text is returned. This check is done on the first character. If the first character is not equal to greater than 64 and less than 91 and is not greater than 96 and less than 123 then an error is reported. The error is reported to the listing file and an empty text equal to “” is returned. Neither of these errors is fatal. The *getOperation()* checks for only one error. This check is very simple. The input record line needs to be at least a length of 10 in order to have an operation present. An ‘If’ statement checks the length of the line. If the line is not long enough, then an error is reported to the listing file and an empty operation text is returned from the operation. This error is not fatal.

Group III: Errors Caught in Pass Two

Pass two checks for and reports less errors than pass one. The first error it checks for is for every record within the *interTable*. Pass two checks if the operation code for each of those records is in the machine table. If the operation length is equal to zero then an error is printed to the listing file. This error is also fatal, and pass two is broken from. The next check is done if the operation is in the pseudo operation table. The next two error checks are very similar. If the operation is either ‘NMD’ or ‘RES’, then the operand needs to represent an integer. An ‘If” statement check to see whether or not the record’s operand represents an integer. If it does not then an error is reported by printing it to the listing file. This error is not fatal in either case.

Data Element Dictionary

