Introduction

This program, the portfolio project for the class, is the final step up in difficulty. Remember, grace days cannot be used on this assignment. Once again the Rubric (see below) now has a number of point deductions for not meeting requirements. It it not uncommon for a student to generate a program that meets the **Program Description** but violates several **Program Requirements**, causing a *significant* loss in points. Please carefully review the Rubric to avoid this circumstance.

The purpose of this assignment is to reinforce concepts related to string primitive instructions and macros (CLO 3, 4, 5).

1. Designing, implementing, and calling *low-level I/O procedures*
2. Implementing and using *macros*

What you must do

Program Description

We had an intern working on our thermometer data storage and he messed it up. The readings are good, but the order is reversed. We need you to write us a efficient little MASM program to read in the temperature measurements from a file, and print them out with their order corrected! This program must perform the following tasks (as always, be sure to check the Requirements section):

- Implement and test three **macros** for I/O. These macros should use Irvine's `ReadString` to get input from the user, and `WriteString` and `WriteChar` procedures to display output.
  - `mGetString`: Display a prompt *(input, reference*), then get the user's keyboard input into a memory location *(output, reference*). You may also need to provide a count *(input, value)* for the length of input string you can accommodate and provide a number of bytes read (*output, value)* by the macro.
  - `mDisplayString`: Print the string which is stored in a specified memory location *(input, reference*).
  - `mDisplayChar`: Print an ASCII-formatted character which is provided as an immediate or constant (*input; immediate, constant, or register)*.
- Implement and test the following two **procedures** which use string primitive instructions
  - `ParseTempsFromString`: {parameters: *fileBuffer* (reference, input), *tempArray* (reference, output)}
    1. fileBuffer will contain a number (`TEMPS_PER_DAY` (*CONSTANT*)) of string-formatted integer values, separated by a delimiter. The `DELIMITER` must be defined as a character CONSTANT so that we can change it during testing.
       - `TEMPS_PER_DAY` should be initially set to 24
       - `DELIMITER` should be initially set to the comma *character* `,`
    2. Convert (using string primitives) the string of ascii-formatted numbers to their numeric value representations. Some values will be negative and others will be positive.
    3. Store the converted temperatures in an SDWORD array (*output parameter, by reference*).
  - `WriteTempsReverse`: {parameters: *tempArray* (reference, input)}
    1. Print an SDWORD integer array to the screen, separated by a CONSTANT-defined `DELIMITER` character.
    2. The integers must be printed in the reverse order that they are stored in the array.
    3. Invoke the `mDisplayChar` macro to print the `DELIMITER` character.
- Write a test program (in `main`) which uses the `ParseTempsFromString` and `WriteTempsReverse` procedures above to:
  1. Invoke the `mGetString` macro (see parameter requirements above) to get a file name from the user.
  2. Open this file and read the contents into a file buffer (`BYTE` array). File formatting follows...
     - The file will contain a series of positive or negative ASCII-format integers, separated by a `DELIMITER`.
     - Each line of numbers will have `TEMPS_PER_DAY` values.
     - The last number of each line also has a `DELIMITER` after it.
  3. Use `ParseTempsFromString` to parse the *first line* of temperature readings, convert them from ASCII to numeric value, and store the numeric values in an array.
  4. Use `WriteTempsReverse` to print the temperature values in the reverse order that they were stored in the file (print to the terminal window). Crazy interns!

Program Requirements

1. `WriteInt` may be used to print the SDWORD array.
2. `mDisplayString` must be used to display all strings.
3. Conversion routines **must** appropriately use the `LODSB` and/or `STOSB` operators for dealing with strings.
4. You may not use `ParseInteger32` or other similar pre-written procedures to parse the temperatures.
5. All procedure parameters **must** be passed on the runtime stack using the **STDCall** calling convention (see Module 7, Exploration 1 - Passing Parameters on the Stack). Strings also **must** be passed by reference.
6. Prompts, identifying strings, and other memory locations **must** be passed by address to the macros.
7. Used registers **must** be saved and restored by the called procedures and macros.
8. The stack frame **must** be cleaned up by the **called** procedure.
9. Procedures (except `main`) **must not** reference data segment variables by name. There is a **significant** penalty attached to violations of this rule. Some global constants (properly defined using EQU, =, or TEXTEQU and not redefined) are allowed. These **must** fit the proper role of a constant in a program (nominally static values used throughout a program which may dictate its execution method, similar to `MIN_TEMP` and `MAX_TEMP` in Project 5).
10. The program **must** use *Register Indirect* addressing or string primitives (e.g. STOSD) for integer (SDWORD) array elements, and *Base+Offset* addressing for accessing parameters on the runtime stack.
11. Procedures **may** use local variables when appropriate.
12. The program **must** be fully documented and laid out according to the CS271 Style Guide. This includes a complete header block for identification, description, etc., a comment outline to explain each section of code, and proper procedure headers/documentation.

Notes

1. Reading from files may be new to you. Refer to the CS271 Irvine Procedure Reference document for specifics. Part of advancing as a programmer is using documentation to learn how to do new things... but I'll give a bit of guidance here. The general process of file I/O in MASM is as follows.
   1. Get File Name
   2. Use Irvine OpenInputFile with the filename to open a file. A file must be "open" before you can read from (or write to) it. This procedure will give you a "fileHandle" which is an identifier that your program will use to perform any file I/O.
   3. Check for any possible errors that occurred in opening the file. If there were errors, print an error message and quit.
   4. Use ReadFromFile or WriteToFile appropriately. If reading from the file, ensure your file buffer is large enough to accommodate everything you read from the file.
   5. Once you're finished with reading or writing, close the file.
   6. Proceed with the rest of your program.
2. For this assignment you are allowed to assume each temperature value is between -100 and 200.
3. We will be testing this program with positive ***and*** negative values, and with several single-character delimiters, including commas and spaces.

4. Check the Course Syllabus for late submission guidelines.
5. Find the assembly language instruction syntax and help in the CS271 Instructions Reference.
6. To create, assemble, run, and modify your program, follow the instructions on the course Syllabus Page's "Tools" tab.

Resources

Additional resources for this assignment

- Project Shell with Template.asm
- CS271 Style Guide
- CS271 Instructions Reference
- CS271 Irvine Procedure Reference

What to turn in

Turn in a single .asm file (the actual Assembly Language Program file, not the Visual Studio solution file). File must be named "Proj6_ONID.asm" where ONID is your ONID username. Failure to name files according to this convention may result in reduced scores (or ungraded work).

Example Execution

In this example, user input is shown in ***boldface italics*** for clarity.

Example text file is available here: Temps090124.txt

```
Welcome to the intern error-corrector! I'll read a ','-delimited file storing a series of temperature values.
The file must be ASCII-formatted. I'll then reverse the ordering and provide the corrected temperature
ordering as a printout!
Enter the name of the file to be read: Temps090124.txt

Here's the corrected temperature order!
-1,+2,+5,+10,+15,+20,+25,+30,+34,+38,+42,+45,+40,+35,+30,+25,+20,+15,+10,+7,+3,+0,-2,-3,

Hope that helps resolve the issue, goodbye!
Press any key to continue . . .
```

Extra Credit (Original Project Definition must be Fulfilled)

To receive points for any extra credit options, you **must** add one print statement to your program output **per extra credit** which describes the extra credit you chose to work on. You ***will not receive extra credit points*** unless you do this. The statement must be formatted as follows...

```
--Program Intro--
**EC: DESCRIPTION
--Program prompts, etc—
```

For example, for extra credit option #2, program execution would look like this:

```
Welcome to the intern error-corrector! I'll read a ','-delimited file storing a series of temperature values.
The file must be ASCII-formatted. I'll then reverse the ordering and provide the corrected temperature
ordering as a printout!
*EC: This program implements a WriteVal procedure to convert integers to strings and display them, rather than
using WriteDec/WriteInt.

Enter the name of the file to be read: Temps090124.txt
...
```

Extra Credit Options

1. Convert the program to handle multiple-line input files. Each line is terminated with a Carriage Return character, followed by a Line Feed character. Each input line should correspond to one output line (e.g. "Reversed Input Line 1" becomes "Corrected Input Line 1", "Reversed Input Line 2" becomes "Corrected Input Line 2", etc...) (4pts)
2. Implement a procedure `WriteVal` which receives as an input a positive or negative integer value (input parameter, by value) and converts this value to an ASCII-formatted string representation, then invoke the `mDisplayString` macro to write this string to the terminal window. This new `WriteVal` must be used when printing integers in the program (replaces WriteDec/WriteInt). (4pts)