Assign-02 : Writing a Linux Utility (***encodeInput***)

Description

This assignment has you writing a different utility for Linux. This utility (officially called a filter in UNIX / Linux) could be used to take an ***assembled*** program (e.g. for the Motorola 6808 Microcontroller) and generate something known as an ***S19 download file*** in order to download the assembled code to the actual embedded device. This is similar to your ARM development environment in the MES course – you write and compile your code on a host computer and download it to the ARM board …

This application will encourage you to learn more about the common application-programming model in UNIX / Linux. The utility (or filter) that you create will take ***ANY binary input file***, and transform it into its equivalent S-Record output file, **OR** an assembly file for use in an embedded software development environment. The two different output file formats which are generated by this filter will both be ASCII (human readable).

Objectives

- Reinforce knowledge of binary arithmetic and the hexadecimal numbering system
- Reinforce knowledge of File I/O (both ASCII and binary) in programming – using actual files as well as STDIN and STDOUT
- Practice writing a utility that can use command-shell *redirecting* and/or *piping*
- Practice writing a utility that requires and uses command-line arguments

Assign-02 : Writing a Linux Utility (***encodeInput***)

Background

The S-RECORD Output of your ***encodeInput*** Filter

S-Records are often used in embedded development environments in terms of transferring data from one system to another. Development tools like EPROM burners for example accept data in this format. As you may recall from DEF, an ***ASM*** file (assembly file) is used by embedded development tools to allow you to specify both machine codes and data (like strings) for your target system.

The *Motorola S-Record File* format was developed a number of years ago. It is also known as the ***SREC*** or ***S19*** format. It was developed in the 1970's by Motorola (for the 6800 microprocessor) to allow you to encode your binary files (e.g. your executables) into an ASCII format file for easy downloading to an embedded system. Visit the link above to learn more about the structure of the S-Record.

Example:

Input data:

```
ABCDEFGHIJKLMNOPQRST
```

SREC Output data:

```
S00700005345414ED1
S113000041424344454647484 94A4B4C4D4E4F5064
S1070010515253549E
S5030002FA
S9030000FC
```

As you can see, the S19 (or SREC) file is made up of a number of "S" type records (`S0`, `S1`, `S5` and `S9` in the above example).
The general format of each of these "S" records is:

TTCCAAAADDD .. DDMM

Where
- TT – 2 characters indicating the S-Record type
- CC – 2 characters representing a single hexadecimal byte telling how many hexadecimal coded bytes follow in that line
- AAAA - 4 characters representing a hexadecimal address where the data on this S-record line needs to be loaded into memory on the embedded device
- DD..DD - up to 32 characters – representing up to 16 hexadecimal byte values representing the input data
- MM - 2 characters – representing a single hexadecimal value which acts as the checksum value for that S Record

Things to Note

- The ***COUNT*** field (CC) of the specific S Record contains the number of bytes including the *address field* (AAAA), the *data fields* (DD..DD) and the *checksum field* (MM)
- In our filter we will *only be using 4 character address fields (representing a 2 hexadecimal byte address)* – this means you only need to use the S1 record for your data.
  - Since each S1 record is limited to containing a maximum of 16 bytes worth of data – this means that the address value contained in this field will always increment by 16

  - This means that each S1 record will only represent 16 bytes worth of encode data. So if the assembled program that you are trying to encode for downloading is 40 bytes in length – then your resultant file will contain two S1 records representing 16 bytes of data each and one S1 record containing the remaining 8 bytes of data (16+16+8=40)
- The ***CHECKSUM*** field (MM) value is calculated by taking the least significant byte of the ***1's Complement*** value of the **sum** of the COUNT, ADDRESS and DATA fields of the record
  - To calculate this value add the COUNT field's value with the ADDRESS field's value with each of the DATA field's values to get a sum value
  - Then take the 1's Complement (remember DEF) of this sum
  - Then strip off the value in the least significant byte of the resultant value and encode it
- One thing that will strike you as odd about the above coding and sample is for example how I say that the COUNT field is 2 characters in output, but represents a single hexadecimal byte
  - For example the above COUNT field is set to "13" which represents the hex number 0x13 (19 decimal)
  - You will remember from DEF that a value of 0x13 hex could definitely be stored in a single byte of output – so why then does the resultant S Record output use 2 bytes to store it … it stores the character "1" in one output byte and the character "3" in another byte
  - This is a form of hexadecimal coded values that the S Record file uses … remember that the S Record out is always human readable
    - If the output actually stored the value 0x13 in a single byte – the file wouldn't be readable – check out the ASCII table
    - 19 decimal in the ASCII table is known as a character called "device control 3 (DC3)" which is unprintable / unreadable by a human
  - So the hexadecimal encoding that is really happening within this utility is:
    - If the value that needs to be output would end up being 0xA4 as a hexadecimal value then the output will contain "A4" (ASCII code 65 followed by ASCII code 52) instead
    - if the value 9 hexadecimal needs to be output by the program, then the output will contain "09" (ASCII code 48 followed by ASCII code 57)
    - as you can see – each single hexadecimal byte value is translated (encoded) as 2 ASCII output characters
- The S0 record is known as the *header record* of the output file
  - In this utility, I want you to encode your first name in the DATA field of the S0 record
    - e.g. the data fields in the above example "`5345414E`" actually spells SEAN – $53_{16}=83_{10}$=ASCII code for "S"
- The S5 record is found immediately after the last S1 record serves as *summation record* within the output file
  - The AAAA (address field) of this record represents the total number of S1 records that preceded this record in the out
- The S9 record is the *trailer record* in the output file
  - The AAAA (address field) of this record represents the address in memory on the target embedded device where the program actually starts

Assign-02 : Writing a Linux Utility (***encodeInput***)

The ASSEMBLY FILE Output of your ***encodeInput*** Filter

The required *Assembly File* format that your utility can also produce is much more straightforward in its coding than the S-Record format. Essentially, this format translates each byte of input data into a **define constant byte** (`dc.b`) assembly instruction. Each line of the resultant file is allowed to contain a maximum of 16 bytes of input data.

Example:

Input data:

ABCDEFGHIJKLMNOPQRST

Assembly File Output data:

```
dc.b    $41, $42, $43, $44, $45, $46, $47, $48, $49, $4A, $4B, $4C, $4D, $4E, $4F, $50
dc.b    $51, $52, $53, $54
```

Your Task

Create a **C program** that has 4 ***optional switches*** (in C, you might consider these as *command line arguments*). With the exception of the *help* switch, your filter program (called **encodeInput**) will take and interpret input as binary values, and produce ASCII readable output based on these switches. The data values encoded in the output will always be the *hexadecimal representation* of the ASCII value of the input byte.

The switches that **encodeInput** must support are:

- `-i`*`INPUTFILENAME`*
  - This option tells your software the name of the input file.
  - If this file is not specified, your program will obtain input data from the standard input (***stdin*** file handle)
  - If the file is specified and does not exist, your program will present an error
- `-o`*`OUTPUTFILENAME`*
  - This option tells your software the name of the output file.
  - If this file is not specified, and an input file is specified
    - If the `-srec` option is not present, then the output filename will be the input filename, with "`.asm`" file extension appended
      - e.g. `encodeInput -iBinary-01.bin` will automatically create an output file called `Binary-01.bin.asm`
    - If the `-srec` option is present, then the output filename will be the input filename, with "`.srec`" appended
      - e.g. `encodeInput -srec -iBinary-01.bin` will automatically create an output file called `Binary-01.bin.srec`
    - Notice in this assignment, we are not replacing the input file's extension – we are simply appending the output file's extension
  - If this file is not specified and no input file is specified, then all output will be sent to the standard output (***stdout*** file handle)
  - If this file is specified and for some reason, it cannot be open for writing purposes, the filter will produce an error
- `-srec`
  - This option tells your software to output in the S-Record format
  - Without this option specified on the command line, then an Assembly File output will result
- `-h`
  - This option will cause the program to output help information (or *usage statement*) and exit
  - The usage statement is simply the name of the program and its allowable runtime switches

If any other (invalid) switches are present on the command line, the filter will display the usage statement and exit.

Because **encodeInput** uses run-time switches, this program **does not need to prompt the end user for any other kinds of input**. Instead, if written correctly, the end user will be able to redirect or pipe textual data into the application in place of using a file!

Although it probably doesn't need to be said – the only way that **encodeInput** should allow an input stream to end is when it reaches the *end-of-file*. That is, if you are running **encodeInput** and reading from a file, obviously the input ends when it reaches the *end-of-file*. Similarly when you are running the utility interactively and are entering input via the keyboard, the input ends when it reaches the *end-of-file* - not an ENTER key, and not any other special character sequence – just an *end-of-file*. What you need to do is to simulate an *end-of-file* from the keyboard – this is done by entering a **CTRL-D** (pressing the CONTROL key and the D key simultaneously).

Please note that **CTRL-D only works to end the file when entered at the beginning of the input line** – if it is not the first character of the input, then CTRL-D serves to flush (or push) the data ahead of it into the program. For example, ***assume the "D" in the following examples represents when I press CTRL-D and "E" is when I hit enter*** – so if you enter "123D", the CTRL-D simply causes the program to read "123". But if my input is "123ED" – then the CTRL-D is the first character of an input line (because it follows an ENTER) – so this will be interpreted as an EOF character in your program.

Some examples of valid command lines:

```
encodeInput -imyData.dat -omyData.srec -srec
```

- The above will convert `myData.dat` to `myData.srec` as an S-Record file

```
encodeInput -h
```

- The above will output help (usage statement) information and exit

```
encodeInput -omyData.asm
```

- The above will convert standard input into `myData.asm` as an assembly file

```
encodeInput -srec -imyData.dat
```

- The above will convert data from `myData.dat` into `myData.dat.srec` as an S-Record file

```
encodeInput
```

- The above will convert standard input into Assembly formatted standard output

```
ls -l | encodeInput -odirectory.srec -srec
```

- The above will take the piped standard input from `ls -l` command and format as an S-Record file output in `directory.srec`

NOTE: The runtime switches can appear in any order on the command line

Assign-02 : Writing a Linux Utility (***encodeInput***)

OTHER PROGRAMMING NOTES:

- It is expected that your final solution and source code for this utility consists of at least 3 source modules and at least one header file
  - As before – I am asking you to do this to start you thinking about designing your solutions to properly modularize your solution approach into different source code modules (files)
  - Remember that it is good design practice to not place any ***problem-domain*** knowledge in the `main()` function
  - Also it is good design practice to not place any of your solution's functions in the same source module as the `main()` function
- You must comply with SET Coding Standards
  - Make sure to comment your source code appropriately – this includes file header comments, function header comments as well as inline comments
- Your solution structure must include a makefile and also follow the recommended Linux development directory structure as outlined in the Linux-Development-Project-Code-Structure document within eConestoga

Hand in

- Please clean and hand in your encodeInput solution – tar up your solution directory into a file titled "`lastName-firstInitial.tar`" (e.g. John Smith would submit `smith-j.tar`) and submit to the dropbox

Sample BINARY Input File

- I have included a sample binary input file (found in the `A02-Sample-Binary.tar` file included with this assignment). The SREC encoded output file is also included with this sample. Just be aware that the name in the `S0` (header) record in the SREC output is mine.
- If you would like to see the bytes in the sample binary input file – you can use the **od** command as I have mentioned in class. Specifically you enter the following command: `od -x --endian=big binary-input-file.bin`
  - which will dump the contents of the file out in *hexadecimal* format with no byte-swapping (i.e. no little endian-ness to the output)
- Or if you'd prefer to see the binary file as a sequence of single hexadecimal bytes – you can use the following comment (this is my preference) `od –t x1 binary-input-file.bin`
- In Linux you also have the ability to use the *hexdump* command as follows: `hd binary-input-file.bin` which shows you the content in both hexadecimal and ASCII format