# Introduction

Assembly language requires that you be willing to go inside the program files you are using and understand what is happening at the most fundamental level.

You need to know your bits and bytes! You also need to know your hex...

This lab is intended to give you an opportunity to start looking at file internals using a Hex Editor.

Make notes in your log book throughout, so that you can quickly and easily review your work.

## Objectives:

-to observe text file contents using cat and type

-to examine text file contents

-to be able to identify the hex value associated with a particular character

-to understand how offset values work

-to edit file contents

-to observe the different interpretations available for file contents

-to understand the difference between big endian and little endian representations

-to practise with big endian and little endian interpretation

# Preparation

Download and expand AssemblyWeek1 the unit’s SOL page.

## Step 1

**Text files** are files that can be opened and examined meaningfully in a text editor such as notepad and notepad++

**Notes:**

Text files contain visible characters:

upper case and lower-case letters and numbers plus various punctuation characters and whitespace characters including:

space, tab and newline

as well as other more obscure characters, control characters and 'fossil' characters

**Binary files** are files containing values that do not display meaningfully as text.

Run **notepad** and open AssemblyWeek1\HelloWorld.exe (easiest way is open notepad then drag the file into it from Windows explorer)

What do you see?

## Step 2

Open a cmd window and navigate to the **AssemblyWeek1** directory.

Press Windows Key+R to bring up run dialogue box, then enter **cmd**

To easily move to the directory, enter **CD** followed by a space in the cmd window, then drag across the folder icon from windows explorer into the cmd window.

Here you will find samlinux.txt and samwindows.txt. Both text files were created on their respective systems.

Now try:

### type samlinux.txt

**type samwindows.txt**

How do the results compare?

Now try opening both files in Notepad (NOT notepad++)

Now how do the results compare?

## Step 3

Ref: Lab 0.pdf in Notes folder, page 1 : Looking at File Internals with the Bless Editor

A hex editor is a utility that allows you to open, display and change characters or bytes inside

### any type of file

### We will be using an online hex editor for portability. Open Chrome and go to <https://hexed.it/>

Now open samlinux.txt and samwindows.txt in hexed

The display pane is divided into three parts:

**Offset column**

contains the offset (distance) from the start of the file for the first byte displayed displayed in hex!

**Hex display column Char display column**

visible text chars are displayed as text, non-displayable chars as '.' Study the two file displays and make a note of any differences...

What is the hex value for a full stop character?

What is the hex value/values? for the EOL (end of line) character? How does this differ in the samwindows.txt and samlinux.txt file?

**Notes:**

Even with so-called standard text files, there can be differences in the way a file is represented

When you take the samlinux.txt file and load it into the Windows Notepad editor, Notepad is not able to interpret the end of line correctly

Note that if you select a byte by clicking on it in the hex or char view, at the right of the page it will tell you which byte you have selected (e.g. ‘Current Address:0x0000000B)

what is the offset value for the character 's' in samwindows.txt?

Note the values given at the right of the Hexed window:

check this across the different files you have used

and explain the meaning of the value given in the position **0x08**

Copy samwindows.txt to test.txt in Windows explorer.

Edit your test.txt file within Hexed so that it will read as follows:

Sam was a

woman.

You will need insert some bytes into the file by right clicking where you want to insert them and selecting “Insert Bytes Here”

Export the file and check the contents using notepad to confirm that you have done it correctly

## Step 5

Ref: Lab 0.pdf in Notes folder, page 5 : Interpreting Raw Data

Both samlinux.txt and samwindows.txt begin with 'S' which is shown in hex as 53

Working on samlinux.txt, put the cursor in this first character position

Now look at the numeric values given in the Data Inspector (Little-Endian) on the left side of the Hexed window.Note the following values and identify the base being used to display each value:

Unsigned 8 bit integer:

is given in base:

Unsigned 16 bit integer:

is given in base:

**Notes:**

Hexed takes the file it has been given and

displays the bytes within that file as a sequence of hex pairs and shows you the different ways in which those bytes may be interpreted

Remember, Hexed can be used to examine any type of file, not just text files.

For example, you may wish to look at the contents of an assembled file.

The way it is interpreted is up to you!

## Step 6

Ref: Lab 0.pdf in Notes folder, page 6 : Endianness

You will have noticed that some of the interpretations dealt with one byte, whilst others dealt with two or four bytes.

In each case, the sequence of bytes being interpreted begins at the cursor and goes to the right. Now make sure that you open up the section Data Inspector (Big-endian) as well.

You might want to reduce the amount of data visible in these panels. If you click on Settings, then under ‘Visible Data Types’ uncheck everything that isn’t an integer.

Do you open your hard boiled eggs at the big end first or the little end first?

What is the range of numbers (in decimal) that can be represented using 8 bits? How many different (unique) numbers is it possible to represent using 8 bits? What is the range of numbers (in decimal) that can be represented using 16 bits? How many different (unique) numbers is it possible to represent using 16 bits?

**Notes:**

If you want to represent numbers that require more than one byte, the order of the bytes becomes crucial!

Go back to samlinux.txt

The first 8-bit value or byte is 0x53 (83 in decimal)

If the first two bytes were interpreted as an unsigned 16 bit value we might assume it would be

0x5361 (21345 in decimal)

But, it is not that simple...

What is the value in the box labelled 'Unsigned 16 bit'?

**Notes:**

We read numbers from left to right.

eg. 4321, reads as four thousand, three hundred and twenty one

But, when we evaluate numbers: 61+

25

we work from right to left!

Thus, it is important to know which of the two bytes that make up the 16-bit value is the most significant byte and which is the least significant byte

should it be 53 61h or 61 53h?

i.e. are we going to use big endian or little endian ordering?

In fact, this choice is 'baked right into the silicon of the CPU’

A computer architecture that stores the **least significant byte** of a **multibyte value**

at the **lowest offset** is called **little endian** – in this instance it would be stored as 61 53h (little end of the number comes first in little endian)

A computer architecture that stores the **least significant byte** of a **multibyte value**

at the **highest offset** is called **big endian** – in this instance it would be stored as 53 61h (big end of the number comes first in big endian)

Use Hexed to answer the following questions:

You will also find it useful to use the Desktop calculator, in programming view

What is the unsigned 16-bit little endian decimal equivalent of the first two bytes of the file?

What is the unsigned 16-bit big endian decimal equivalent of the first two bytes of the file?

What is the unsigned 32-bit little endian decimal equivalent of the first four bytes of the file?

What is the unsigned 32-bit big endian decimal equivalent of the first four bytes of the file?

If you program in higher level languages such as C, issues such as endianness are normally hidden, unless you are trying to share binary data between different systems. A good example of this is if you have two computers trying to talk via network sockets.

If you program in assembly language, you need to pay attention to endianness at all times.

When you later inspect numeric values stored in memory using a debugger, the same rules apply. What does linux use?

Both, it is independent of the processor implementation.

What matters is the hardware architecture...

| **Processor** | **Endianness** |
| --- | --- |
| Motorola 68000 | Big Endian |
| PowerPC (PPC) | Big Endian |
| Sun Sparc | Big Endian |
| IBM S/390 | Big Endian |
| Intel x86 (32 bit) | Little Endian |
| Intel x86\_64 (64 bit) | Little Endian |
| Dec VAX | Little Endian |
| Alpha | Bi (Big/Little) Endian |
| ARM | Bi (Big/Little) Endian |
| IA-64 (64 bit) | Bi (Big/Little) Endian |
| MIPS | Bi (Big/Little) Endian |

## Step 7

Try these examples:

First convert each of the decimal values to hex

Next, show how they would be stored for a big endian system and a little endian system in 32bits.

254, in hex:

big endian representation:

little endian representation:

158, in hex:

big endian representation:

little endian representation:

2540, in hex:

big endian representation:

little endian representation:

3427, in hex:

big endian representation:

little endian representation:

11223344, in hex:

big endian representation:

little endian representation:

67123 in hex:

bit endian representation:

little endian representation:

24345464, in hex:

big endian representation:

little endian representation:

Further examples:

Take the following hex values and give their unsigned decimal equivalent assuming they are interpreted as big endian and then interpreted as little endian

16-bit Numbers

7d

decimal equivalent assuming big endian: decimal equivalent assuming little endian:

88

decimal equivalent assuming big endian: decimal equivalent assuming little endian:

3a4 (think carefully about this one!) decimal equivalent assuming big endian: decimal equivalent assuming little endian:

f0e (and this one!)

decimal equivalent assuming big endian: decimal equivalent assuming little endian:

1234

decimal equivalent assuming big endian: decimal equivalent assuming little endian:

2a3b

decimal equivalent assuming big endian: decimal equivalent assuming little endian:

32 Bit Numbers:

7654eb

decimal equivalent assuming big endian: decimal equivalent assuming little endian:

70fad07

decimal equivalent assuming big endian: decimal equivalent assuming little endian:

What have you found difficult in today's lab session?

Pay attention to your first, instinctive, response when you find something challenging... Remember that when something is new it often feels uncomfortable

Time spent practicing will make a big difference to your ability to learn new concepts and be able to use them effectively...