

Lab 3

[Submit Assignment](#)

Due Mar 1 by 12:30pm **Points** 100 **Submitting** a file upload **File Types** zip
Available Feb 13 at 1:30pm - Mar 1 at 12:30pm 16 days

Purpose

This lab will give you practice using the many features of the C# programming language, including typecasting, reading binary files, writing text files, bitwise operators, interfaces, abstract classes, inheritance, and integer arithmetic.

Assignment

You will be writing a C# program that reads from a binary file and writes to a text file. The purpose of the program is to read a binary file that encodes numbers in BCD (Binary Coded Decimal) or DPD (Densely Packed Decimal) format. Each value is encoded using 5 bytes. The format of these 5 bytes is described below.

Requirements

1. You must use the C# programming language to solve this problem.
2. You must use the ICompValue interface that BCD and DPD will implement. The interface methods are described below. You must also use good object oriented programming techniques. If you find that you implement the same function for two classes, you must implement an intermediate, abstract class and NOT duplicate code.
3. You must implement two classes: one for BCD and one for DPD. The BCD class must always store a value in binary coded decimal. The DPD class must store a value in densely packed decimal format.
4. The user will specify the input and output files using command line arguments.
5. Each class or interface must be in its own .cs file, and the name of the file must be the same name as the interface or class. For example, if you made a Foo interface, it must be defined in Foo.cs.
6. You must store each number read from the input file into a generic container.
7. You must use the container's Sort function. Since CompValue inherits the IComparable interface, this function will work without any arguments!
8. You must use the .ForEach() function to store each value in your container class to the output file. The output file will be written in plain text, one integer per line.
9. Treat the BCD and DPD as independent classes, meaning DPD may not call any BCD functions and BCD may not call any DPD functions.

[ICompValue Interface]

You must implement the following public interface for your BCD and DPD classes:

```
interface ICompValue : System.IComparable {
    /// System.IComparable.CompareTo(object)
    /// Needs to:
    /// return -1 if this.Val < object.Val (object typecasted into ICompValue)
    /// return 0 if this.Val == object.Val (object typecasted into ICompValue)
    /// return 1 if this.Val > object.Val

    /// Raw property (gets/sets the RAW DPD or BCD)
    uint Raw { get; set; }
    /// Decoded property (gets the actual number)
    uint Val { get; }
}
```

You will need to create an abstract class that implements this interface to avoid duplicating code that is the same for both BCD and DPD classes and that manipulates an instantiated member variable. Then, your BCD and DPD classes will extend the abstract class.

Input File

The input file will contain a list of values, 5 bytes at a time. **All of the values are in big-endian format (most significant bit comes first).**

The first byte is either a 0 or 1. If the byte is a 0, the remaining 4 bytes are a BCD encoded value. If the byte is a 1, the remaining 4 bytes are a DPD encoded value. Both BCD and DPD encoded values may be present in the same file.

Binary Coded Decimal Format

The binary coded decimal format uses 4 bits to encode a decimal value from 0 - 9. Since we're encoding using 4 bytes, that means that each BCD may encode up to 8 decimal digits (4 bytes contains 8 groups of 4 bits).

An example of a BCD value would be 9531. This will be encoded using 4 bits per decimal value. So, the BCD encoding would be 1001 0101 0011 0001. In binary, 1001 is 9, 0101 is 5, 0011 is 3, and 0001 is 1.

The input file will use 4 bytes for each BCD encoding, meaning that each BCD encoding is 8 digits.

Example: Take 8 digits: 12345678. All you need to do is take each individual digit and encode it into binary:

0001	0010	0011	0100	0101	0110	0111	1000
1	2	3	4	5	6	7	8

Densely Packed Decimal Format

The densely packed decimal format uses 10 bits for 3 decimal digits. Since we're using 32 bits (4 bytes), the upper two bits are not used and should be ignored. This also means that 30-bits will be used to store 9 decimal digits. Notice that this is one more than BCD, hence the name "densely packed decimal". The more space given for the format, the more efficient this DPD becomes over the BCD format.

The 10-bit format stores up to 3 digits, and there are 8 different encodings based on the values that need to be encoded:

DPD Format Encodings

BCD-unpacked format (12-bits)	DPD-packed format (10-bits)	Digit Range
0abc 0def 0ghi	abc def 0 ghi	0-7 0-7 0-7
0abc 0def 100i	abc def 100 i	0-7 0-7 8-9
0abc 100f 0ghi	abc ghf 101 i	0-7 8-9 0-7
100c 0def 0ghi	ghc def 110 i	8-9 0-7 0-7
100c 100f 0ghi	ghc 00 f 111 i	8-9 8-9 0-7
100c 0def 100i	dec 01 f 111 i	8-9 0-7 8-9
0abc 100f 100i	abc 10 f 111 i	0-7 8-9 8-9
100c 100f 100i	00 c 11 f 111 i	8-9 8-9 8-9

This table encodes three decimal digits at a time. For the entire 4-byte sequence, you'll need to encode 3 digits, 3 times for a total of 9 digits.

Example 1: Take the three decimal digits 978.

1. Look in the table to see which row supports 8-9 0-7 8-9, which is the 6th row (dec 01 f 111 i).
2. Convert 978 into the unpacked, 12-bit format: 1001 0111 1000
3. Take the digits needed for the given row: dec, f, and i
4. Substitute the value for dec, f, and i: dec = 111, f = 1, i = 0
5. Write the packed value: dec 01 f 111 i becomes 111 01 1 111 0
6. Repack into 10-bit packed format: 111 011 1110

So, 978 is encoded as 111 011 1110 in DPD format.

Example 2: remember, we can have up to 9 digits, so take an example of 9 digits:

123456789

We take these 3 digits at a time and store the bytes in order:

```

Digits : 12-bit Unpacked : DPD Row : Sequence : 10-bit Packed
123    : 0001 0010 0011 : Row 1 : abc def 0ghi : 001 010 0011
456    : 0100 0101 0110 : Row 1 : abc def 0ghi : 100 101 0110
789    : 0111 1000 1001 : Row 7 : abc 10f 111i : 111 100 1111
          abc def ghi

```

The upper two bits (index 31 and 30) will be always 0 in DPD encoding. Then we put in 123, then 456, then finally 789:

```
00  001 010 0011    100 101 0110    111 100 1111
      [123]          [456]          [789]
```

You can stop there, but I'm going to repack into groups of 4 to make it easy to convert to hex:

```
0000 1010 0011 1001 0101 1011 1100 1111
  0    a    3    9    5    b    c    f

0x0a395bcf
```

So, 123456789 encodes into 0x0a395bcf using 4 bytes in DPD format.


Output File

The output file will be a text file of the sorted, decoded integers, one line at a time. For example, the output file might look like:

```
0
5
7
9
16
```

Testing

The file [test.2000](#) has 2000 random integers encoded in a random mix of both BCD and DPD format. The file [bcd.2000](#) has 2000 random integers encoded in BCD format only. The file [dpd.2000](#) has 2000 random integers encoded in DPD format only.

The output files [test.2000.txt](#) , [bcd.2000.txt](#) , and [dpd.2000.txt](#)  are the output files for the respective input files.

You can check to see if your output is the same as mine by using:

```
diff test.2000.out test.2000.txt
```

If the files are the same, nothing will be printed. Otherwise, diff will show you what is different between the two files.

I recommend you start with BCD, since it is easier. Then test. Then move to DPD.

Plagiarism

Please review the plagiarism policy listed in the [syllabus](#). This lab is an individual effort!

Submission

Zip all .cs files and any other file needed to compile your code into lab3.zip.

Submit lab3.zip