2. Workingenvironment

On the teaching server, **`ictteach.its.utas.edu.au`**, you must make a directory named **`kit213script`** in your home directory and use this directory as your *working directory* for all assignment development.

1. The first time you start working on the assignment, after logging on, type the following commands ($ is the prompt, do not type that):
2. **`$ mkdir kit213script`**
3. **`$ cd kit213script`**
4. Every other time you log on, simply do the following and then continue working:
5. **`$ cd kit213script`**

3. Allowed Syntax/Commands

All scripts must be completed using syntax and commands discussed in past (and some future) practical classes – no additional referential syntax (that is not discussed in practicals) is allowed. This means your script solutions must not use *alternative* constructs and syntax e.g. solutions found through websites via google searches that use commands and shell syntax that we have not covered in the practical content, and you must not use external parties or other individuals to write the scripts for you (this is considered plagiarism and academic misconduct).

Your solution is restricted (i.e **not allowed**) to use the shell "builtin" binary conversion syntax e.g. **`echo $((2#101010101))`**

This restriction also extends to any other command that has not been discussed in classes – the following list shows *some* of the common methods (**but not all**) found through a brief google search – **none are permitted**:

`awk sed printf xxd od perl ibase, obase` , **`bc`**

The unit also has not discussed *arrays*, so syntax similar in form to **`powers=(128 64 32 16 8 4 2 1) #an array`**

```
for i in ${!powers[@]}; do

something ${powers[$i]} done
```

is strictly prohibited.

Instead, your solution must take an iterative (looping) approach to demonstrate your understanding of iteration and selecting individual characters from a string (a string is just a sequence of characters).

5. ScriptingTaskOverview

5.1 Scenario

There is a long tradition of producing line-based character art for a terminal using the limited character representation available in the ASCII character set. In this assessment task, you are working for a fictional company that wants to create some simple command-line “banners” that include letters, numbers and possibly patterns from pre-defined binary data that will need to be converted from decimal format. The pre-defined decimal-format data for the alphabet and numerals has been created by the company’s art department.

Each individual pattern that will ultimately be shown in the terminal window will be based on 8 rows of binary data (an 8 by 8 grid of “pixels”). As an example, consider a representation of the letter ‘A’:

Column value								
128 2^7	64 2^6	32 2^5	16 2^4	8 2^3	4 2^2	2 2^1	1 2^0	**Row Value**
								0
		■	■	■				56
	■	■		■	■			108
■	■				■	■		198
■	■				■	■		198
■	■	■	■	■	■	■		254
■	■				■	■		198
■	■				■	■		198

Each column in the diagram above represents a power of two, and each entire row represents an 8-bit value (a byte), so the whole letter ‘A’ above uses 8 bytes for this representation. If a pixel (a particular row and column location) is white, that is the equivalent of a zero in the column for that row. If a pixel is black, it is the equivalent of a 1. In pure binary, the whole letter above is represented by the following binary patterns:

```
00000000
00111000
01101100
11000110
11000110
11111110
11000110
11000110
```

The value for the top row here is 0 in decimal. The next row has a 1 in the 2^5 column (32), a 1 in the 2^4 column (16), and a 1 in the 2^3 (8) column. This makes the second row (byte) value 32+16+8 = 56, the row value for the third row 64+32+8+4 = 108 etc.

5.2 Provided source “artwork" files

The entire uppercase alphabetic characters and the numeric characters 0-9 have been pre-drawn by the art department in a similar manner to that describe above. However, the data for these characters has unfortunately been generated as a **series of individual files** (with 8 files per character/pattern) that contain no data – instead, the encoded data binary value is part of each file's name as a decimal number.

Continuing with the letter 'A' example as well as the representation for the letters 'B' and 'C' (shown below), they are represented in the provided source files using the following filenames:

Letter 'A'	Letter 'B'	Letter 'C'
`AA1-0.grf`	`AB1-0.grf`	`AC1-0.grf`
`AA2-56.grf`	`AB2-252.grf`	`AC2-60.grf`
`AA3-108.grf`	`AB3-198.grf`	`AC3-102.grf`
`AA4-198.grf`	`AB4-198.grf`	`AC4-192.grf`
`AA5-198.grf`	`AB5-252.grf`	`AC5-192.grf`
`AA6-254.grf`	`AB6-198.grf`	`AC6-192.grf`
`AA7-198.grf`	`AB7-198.grf`	`AC7-102.grf`
`AA8-198.grf`	`AB8-252.grf`	`AC8-60.grf`

Column								
128 2^7	64 2^6	32 2^5	16 2^4	8 2^3	4 2^2	2 2^1	1 2^0	**Row**
								0
■	■	■	■	■	■			252
■	■				■	■		198
■	■				■	■		198
■	■	■	■	■	■			252
■	■				■	■		198
■	■				■	■		198
■	■	■	■	■	■			252

Column								
128 2^7	64 2^6	32 2^5	16 2^4	8 2^3	4 2^2	2 2^1	1 2^0	**Row**
								0
		■	■	■	■			60
	■	■			■	■		102
■	■							192
■	■							192
■	■							192
	■	■			■	■		102
		■	■	■	■			60

5.3 Source artwork original naming convention

0
60
102
192
192
192
102
60

The format of each provided source filename is the following, which is only important if you manually copy any of the files to your own local directory:

First character	**Second character**	**Third character**	**Fourth character**	**Fifth (and possibly sixth and seventh) characters**	**Last 4 characters**
A literal ‘A’	The letter being represented (i.e. ‘A’ , ‘B’ , ‘C’ … ‘Z’, ‘0’ .. ‘9’)	The row number from each 8x8 grid, starting from 1	A dash (-)	The decimal value of the original byte for each row in the character’s representation grid – this ranges from 0 .. 255	A file extension – the source files use **`.grf`**

All provided source art files can be found in the following directory:

```
/units/kit213/assignment
```

Try to list the contents of that directory (with one filename per line using the `-1` (*minus one*) option, (the `$` is the prompt)):

```
$ ls -1 /units/kit213/assignment
```

You will see in the output that the files are listed in a particular order – this is the reason for the filename convention used, as `ls` will list the files in alphabetic and numeric order, preserving the grouping of 8 files per represented character. There are some extra “symbols” that are also included at the end that you can also use/test with if you want to – these include:

- The space character (ZA…)
- A black solid square (ZB…)
- A square shape (ZC…)
- A triangle shape (ZD…)

5.4 Copying source artwork

• A checkerboard (ZE…)
• A diamond shape (ZF…)
• Diagonal left lines (ZG…) • Diagonal right lines (ZH…)

Part of your assignment task will require you to initially copy some of the source artwork files to your own local directory for processing by the script you will be creating later, however, to simplify and aid with this copying, a command-line program is provided, called **copyBinaryArt**. This program will copy all of the 8 source files associated with a particular character (letter or number) or groups of letters and numbers, but during the copying will replace the first 3 letters of the source filenames with an incrementing 3-digit sequence number. This sequence number will preserve the order of the filenames when they are processed and displayed in the terminal window by your script.

`copyBinaryArt` can be run via the following (which will show how it is used): **`$ /units/kit213/assignment/copyBinaryArt`**

```
Usage: copyBinaryArt art-source art-destination starting-sequence-number string
art-source: the source directory containing the original decimal-named files
art-destination: the destination directory to copy the decimal-named files to starting-
sequence-number: starting number used for first file created, incremented for each file
string: the string to be converted to 8-bit decimal-named files
```

For example, to copy the source art files to represent the text “Hi”, the command would be: (assuming you have created a subdirectory in your current directory called “**`myDir`**”)

```
$ /units/kit213/assignment/copyBinaryArt /units/kit213/assignment myDir 1
“HI”

Source directory /units/kit213/assignment verified to exist! Destination
directory myDir verified to exist!
The starting sequence number is 1
The string to convert is: HI

Copying files for the character 'H': Copying files for the character 'I':
```

After the copy has completed, the contents of **`MyDir`** would be:

001-0.grf **002-198.grf** **003-**	*Files that represent the letter "H"*

198.grf 004-198.grf 005-254.grf 006-198.grf 007-198.grf 008-198.grf	

009-0.grf 010-252.grf 011-48.grf 012-48.grf 013-48.grf 014-48.grf 015-48.grf 016-252.grf	*Files that represent the letter "I"*

The ultimate task for your script to complete then is to process all of the files that are contained in your local directory, converting the decimal part of each filename to binary, replacing zeros in the resulting binary values with spaces, and replacing ones with some prominent character (like an X) before displaying the result on the terminal screen. Bear in mind when your script is assessed, the marker will use completely different source files to test your script.

5.4 Source filename validity

Because of poor quality control from the art department, occasionally some files may be present in the local source directory that is used for processing (and thus testing by the marker) that do not correctly conform to the file-naming scheme. Your script will have to correctly process these files and correctly categorise (and move) them to another directory. If you have copied some valid source files to your local directory, then you will also need to manually create some invalidly named files to test your script correctly.

5.4.1 Valid filenames

Filename format: ***XYX-decimal.ext***

➢ You are free to create your own additional artwork (using 8 filenames to represent an individual 8 by 8 grid), but your script (see below) must be able to process any validly named file that includes the naming convention described above i.e. 3-digit sequence number, dash, decimal value (0 up to 255), then a file extension (e.g. .**grf**).

Example: directory

any of the following would be considered potentially valid filenames in your local source (the validity will also depend on which particular filename extension has been specified)

- *`XYX`* : a 3-digit sequence code
- `-` : the dash character
- *`decimal`* : a numeric value between 0 and 255 inclusive
- *`.ext`* :a file extension such as .grf or .doc, .xls etc

Example: any of the following would be considered potentially valid filenames in your local source directory (the validity will also depend on which particular filename extension has been specified)

```
001-65.grf 369-0.txt 501-127.doc 999-255.xls
```

5.4.2 Invalid filenames

Anything not fitting the patterns described above is **invalid**. Your script will need to categorise each invalid source filename as detailed below (in order of priority), and then **copy the invalid source file** to a specific subdirectory of your destination directory (see Task Requirements later).

- INVALID1 - one or more non-numeric characters where the decimal value should be.
- INVALID2 - numeric-only decimal values, but the decimal value is more than 255.
- INVALID3 - one or more non-numeric characters where the sequence code should be.
- INVALID4 - numeric-only sequence codes, but there are more than three digits.
- INVALID5 - numeric-only sequence codes, but there are less than three digits.
- INVALID6 - no sequence code or no decimal part.
- INVALID7 - no file extension, or the extension is different to the argument to the script.

Examples of filenames for each category:

Category	**Example invalid filenames (*assume extension specified is .grf*)**		
INVALID1	001-ba0.grf	001-bad.grf	123-12*.grf
INVALID2	123-256.grf	999-999.grf	111-1234.grf
INVALID3	a-000.grf	01x-111.grf	1q7-127.grf
INVALID4	1222-122.grf	11999-100.grf	99999-254.grf
INVALID5	12-122.grf	1-100.grf	99-254.grf
INVALID6	wrong.grf	-123.grf	123-.grf
INVALID7	badfile	123-100.txt[1]	911-000.doc[1]

6. Task Requirements (this is what your script must do)

1. *Your script for this task* ***must*** *be named* ***`displayBanner.sh`*** *If your script has a different name, it will not be assessed.*
2. Make sure your script is not unnecessarily complex - your script should use **consistent indentation** and include **whitespace/blank lines** (where relevant) to make the script more

logical and easier for a person to read. You must also include **general inline comments** in the script code to indicate the purpose of more complex command combinations etc.

3. Your script must start with the following first line:

   ```
   #!/bin/sh
   ```

   (*this specifies the shell to be used to interpret the rest of the commands in the script*)

4. Your script **must include comments** at the beginning (near the top of the script) to specify:
   1. the script's purpose,
5. Your script **must accept 3 command-line arguments** provided when the script is run:
   1. Argument 1 – the local source directory containing the files to be processed
   2. Argument 2 – the local destination directory that will be used to copy invalid filenames to
   3. Argument 3 – the file extension (without a leading dot) to be used for valid source files (eg **`grf`**)

      The directory names and the file extension that are provided by arguments 1,2 and 3 must not be 'hardcoded' (specified literally) in your script, i.e., their value must come from the command line arguments. If any of the 3 arguments are missing, the script should provide a usage warning message and then exit – **the user must not be prompted to enter missing values when the script is running**. See ***Example Output***.

5. Near the beginning of your script, **you need to check** for each of the directories associated with the arguments 1 and 2 (the source directory and the destination directory):
   1. Exists. If the directory does not exist, your script must display an error message and then it should exit. The directory name provided must use a relative path – see ***Example Output*** for examples.
   2. Is readable, executable (for both directories) and writeable (for the destination).
      - i. If the directory is not readable, your script must display an error message and then it should exit.
      - ii. If the directory is not writeable, your script must display an error message and then it should exit.
      - iii. If the directory is not executable, your script must display an error message and then it should exit.

6. For **every file** in the specified source directory, the script must:

a. For a **validly named** file:

i. Extract the **decimal value** from the filename, **convert it to 8-bit binary**, and then output to the terminal window the binary value but with every 0 replaced with a space character (" ") and every 1 replaced with an "X" character. *The following is an example of the decimal value 59 (00111011$_2$) displayed with the space character shown below as a gray block ▒ for clarity purposes only):*

purposes only):

```
▒▒XXX▒XX
```

b. For an **invalidly named file** that does not match the validity test:
i. display the following output to the terminal screen **after all valid files have**

been processed (i.e defer the output to the end of processing):
• ***X*** refers to the specific invalid category (see section 5.4.2)

- *filename* refers to the original source filename
- *invalid-description* refers to the text defined at the end of this step)

```
INVALIDX - filename (invalid-description)
```

Category	*invalid-description*
INVALID1	non-numeric decimal part
INVALID2	decimal part too large
INVALID3	invalid sequence code - non-numeric characters
INVALID4	invalid sequence code - too many digits
INVALID5	invalid sequence code - too few digits
INVALID6	no sequence code or decimal part
INVALID7	filename wrong or missing/wrong file extension

ii. Move the invalid file to a subdirectory of the local destination directory that was specified as the second argument to the script, and call the subdirectory **INVALID*X*** (the script should only create the subdirectory if it does not already exist and also only when a file in this category is found).

8. If the source directory specified as the first argument to the script contains no **files**: a. display the following output to the terminal screen:

• ***sourcedir*** refers to the directory name specified in argument 1

```
The directory sourcedir contained no files…
```

b. exit the script

b. exit the script

9. If the source directory specified as the first argument to the script contained **files**:
    a. display the following output to the terminal screen as the very last line of output:
        - *filecount* refers to the number of files processed by the script

```
filecount files were processed
```

    b. exit the script

7. Test output for submission

Once your script is ready for submission, you must also include some specific test output. The easiest way to do this is to make a *typescript* (log) of the terminal window output. When ready, create a subdirectory for some specific art files – eg **submitSource**

7.1 Create a new typescript:

```
$ script
Script started, file is typescript
```

7.2 Copy artwork files

Copy the art files for your student number and first and last names to the submitSource directory, for example, if your student number is 123456 and your name is John Smith, you would use:

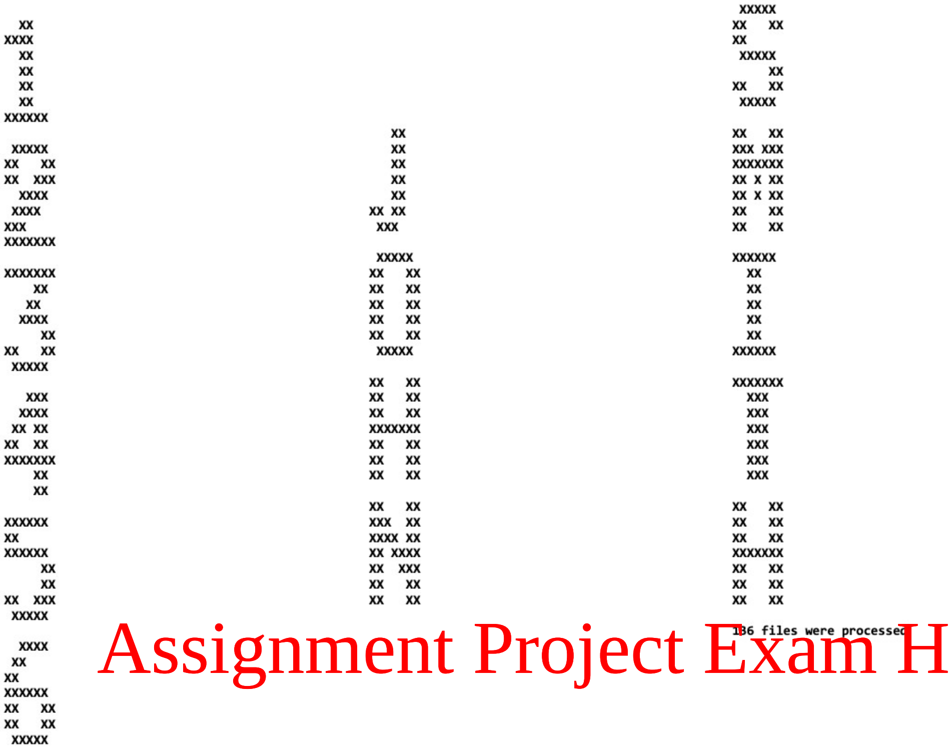
```
$ /units/kit213/assignment/copyBinaryArt /units/kit213/assignment submitSource 100
"123456 John Smith"

Source directory /units/kit213/assignment verified to exist!
Destination directory submitSource verified to exist!
The starting sequence number is 100
The string to convert is:  123456 John Smith
Copying files for the character '1':
Copying files for the character '2':
etc…
```

7.3 Run test output

```
$ ./displayBanner.sh submitSource submitDest grf
```

(output shown in 3 columns for brevity)



```
                                         xxxxx
  xx                                    xx   xx
xxxx                                    xx
  xx                                     xxxxx
  xx                                         xx
  xx                                    xx   xx
  xx                                     xxxxx
xxxxxx
                       xx               xx   xx
 xxxxx                 xx               xxx xxx
xx   xx                xx               xxxxxxx
xx  xxx                xx               xx x xx
  xxxx                 xx               xx x xx
 xxxx               xx xx               xx   xx
xxx                  xxx                xx   xx
xxxxxxx
                     xxxxx              xxxxxx
xxxxxxx             xx   xx               xx
    xx              xx   xx               xx
   xx               xx   xx               xx
  xxxx              xx   xx               xx
     xx             xx   xx               xx
xx   xx              xxxxx              xxxxxx
 xxxxx
                    xx   xx             xxxxxxx
   xxx              xx   xx               xxx
  xxxx              xx   xx               xxx
 xx xx              xxxxxxx               xxx
xx  xx              xx   xx               xxx
xxxxxxx             xx   xx               xxx
    xx              xx   xx               xxx
    xx
                    xx   xx             xx   xx
xxxxxx              xxx  xx             xx   xx
xx                  xxxx xx             xx   xx
xxxxxx              xx xxxx             xxxxxxx
     xx             xx  xxx             xx   xx
     xx             xx   xx             xx   xx
xx  xxx             xx   xx             xx   xx
 xxxxx
                                        136 files were processed
  xxxx
 xx
xx
xxxxxx
xx   xx
xx   xx
 xxxxx
```

7.4 End typescript

End the typescript by pressing control-D on the keyboard:

```
Script done, file is typescript
```

You will then have a file called "typescript" containing the output captured – submit this file.

8. Algorithm Hints

The suggested approach is you take the decimal part from the filename and then iteratively divide the value by decreasing powers of 2 (starting with 128), working out the remainder, and repeating by dividing the next (lower) power of 2 into the remainder (this was shown in a lecture and some practical exercises called *comparison with descending powers of two approach*).

Two expression operators to convert decimal to binary that may be useful:

- expr x / y (*divide x by y*)
- expr x % y (*get the remainder from x divided by y, % is the modulus operator*)

Example – to convert the decimal value 62 to 8-bit binary:

- 1st binary digit = 62 / 128 = **0**; remainder is 62 % 128 = 62
- 2nd binary digit = remainder (62) / 64 = **0**, remainder is 62 % 64 = 62
- 3rd binary digit = remainder (62) / 32 = **1**, remainder is 62 % 32 = 30
- 4th binary digit = remainder (30) / 16 = **1**, remainder is 30 % 16 = 14
- 5th binary digit = remainder (14) / 8 = **1**, remainder is 14 % 8 = 6
- 6th binary digit = remainder (6) / 4 = **1**, remainder is 6 % 4 = 2
- 7th binary digit = remainder (2) / 2 = **1**, remainder is 2 % 2 = 0
- 8th binary digit = remainder (0) / 1 = **0**, remainder = 0 % 1 = 0

The result is 00111110. Your algorithm should implement this but by using iteration (looping).

Another tricky component you may find slightly more challenging is determining which character(s) in a valid filename are part of the decimal part. This is because the decimal part may be 1, 2 or 3 characters long. One suggested approach includes using the **basename** command (introduced here).

basename can strip the extension (called a suffix) from a filename.

You can determine the filename without the file extension/suffix part via:

```
file=some method of getting the current whole filename to be processed…
extention=the extension provided as the 3rd argument to the script
filebase=`basename "$file" .$extension`
```

e.g. if **file** is **123-254.grf**, **extension** is **grf**, then **filebase** would be **123-254**

```
INVALID4 - 9999-128.grf invalid sequence code - too many digits
INVALID3 - a-000.grf invalid sequence code - non-numeric characters
INVALID7 - badfile filename wrong or missing/wrong file extension
INVALID6 - wrong.grf no sequence code or decimal part
18 files were processed
```

The contents of **exampleDest** after the script has run are:

```
      exampleDest /:              exampleDest /INVALID2:     exampleDest /INVALID5:
      INVALID1                    000-256.grf                0-128.grf
      INVALID2                                               02-128.grf
      INVALID3                    exampleDest /INVALID3:
      INVALID4                    a-000.grf                  exampleDest /INVALID6:
      INVALID5                                               wrong.grf
      INVALID6                    exampleDest /INVALID4:
      INVALID7                    9999-128.grf               exampleDest /INVALID7:
                                                             000-255.doc
      exampleDest /INVALID1:                                 badfile
      001-ba0.grf
      001-bad.grf
```

9.3 Examples showing error messages

Missing one or more command-line arguments

```
$ ./displayBanner.sh
Usage: ./displayBanner.sh source-directory destination-directory file-extension
```

Source directory does not exist

```
$ ./displayBanner.sh badSource exampleDest grf
Source directory badSource not found
```

Destination directory does not exist

```
$ ./displayBanner.sh badSource exampleDest grf
Destination directory exampleDest not found
```

Filename extension does not match the majority of files

```
$ ./displayBanner.sh exampleSource exampleDest bad

INVALID7 - 100-0.grf filename wrong or missing/wrong file extension
INVALID7 - 101-48.grf filename wrong or missing/wrong file extension
INVALID7 - 102-240.grf filename wrong or missing/wrong file extension
...
```

[truncated – all files with an extension that does not match will be category INVALID7]

Source directory not readable or executable

```
./displayBanner.sh exampleSource exampleDestination grf
Source directory exampleSource is either not readable or executable
```

Destination directory not readable or executable or writeable

```
$ ./displayBanner.sh exampleSource exampleDestination grf
Destination directory exampleDestination is either not readable, executable or
writable
```

It is then easier to determine the decimal part by using the filename with the extension removed, and other commands (such as **cut**)

9. Example output

9.1 Example files

You of course are free to create any directory you like in your **kit213script** directory and populate it with files for testing purposes (this is highly recommended – you need to test your script works correctly!). The marking process will **not** use the filenames or directory name that are listed below, marking will use a different directory and different testing files.

In this example, assume **kit213script** is the current working directory, and subdirectories called **exampleSource** and **exampleDest** have been created inside **kit213script**. The contents of **exampleSource** are the following (invalid) files:

Filename
000-256.grf
001-ba0.grf
001-bad.grf
0-128.grf
02-128.grf

Filename
9999-128.grf
a-000.grf
badfile
000-255.doc
wrong.grf

The source files for the art for the letter "A" have also been copied to **exampleSource**:

```
$ /units/kit213/assignment/copyBinaryArt /units/kit213/assignment exampleSource 100 "A"
Source directory /units/kit213/assignment verified to exist!
Destination directory exampleSource verified to exist!
The starting sequence number is 100
The string to convert is:  A
Copying files for the character 'A':
```

The following is a sample output of the script you must develop. The text up to and including the **$** is the shell prompt:

9.2 Example of successful run

```
$ ./displayBanner.sh exampleSource exampleDest grf

  XXX
 XX XX
XX   XX
XX   XX
XXXXXXX
XX   XX
XX   XX

INVALID7 - 000-255.doc filename wrong or missing/wrong file extension
INVALID2 - 000-256.grf decimal part too large
INVALID1 - 001-ba0.grf non-numeric decimal part
INVALID1 - 001-bad.grf non-numeric decimal part
INVALID5 - 0-128.grf invalid sequence code - too few digits
INVALID5 - 02-128.grf invalid sequence code - too few digits
```