Assignment: ADIDAS

1 Introduction 2

2 Preliminary exercise 2

3 Adidas Assignment 3

3.1 Background 3

3.2 Principles 4

4.2.1 Nibbles 4

4.2.2 Parity bits 4

3.3 Modules 6

3.3.1 Design 7

3.3.2 Files that need to be implemented 8

3.3.3 Build instructions 8

3.4 Part A: Encoding 9

3.5 Part B: Channel 10

3.6 Part C: Decoding 11

4 Submission 13

# Introduction

This document describes the ADIDAS Assignment. It is the **A**dvanced **D**evelopment for **I**nternet where all **D**ata **A**rrives **S**afely. It is a typical example of an application that you can encounter in embedded devices.

The main learning goals of this assignment are:

* bit manipulations
* modular design of an application
* unit tests

The Adidas Assignment is split into 3 parts (A, B, and C). It is advised to implement them in that order.

The preliminary exercise helps you in understanding the basic functionality of bit manipulations.

# Preliminary exercise

This exercise helps you to learn the bit manipulations in C and will help you to write small methods that you can possibly re-use later on.

In the directory Assignments, go to the directory BitManipulations:

* Implement all methods as given in ‘bit\_stuff.h’. Please look in the ‘bit\_stuff.h’ header file for a description of each method.
* Run the unit tests (via the command line: make test) to make sure everything is properly implemented!

# Adidas Assignment

## Background

When you want to communicate with another system (for example via wired or wireless Internet), you expect that receiver gets the same data as what you have sent:



**"Hello"**



**"**

**Hello"**

But in reality, all kind of interference is possible, for example because of lightning, bad cables, microwaves, or high voltage towers. So the data may be corrupted along the route:



**"Hello"**



**"@effT"**



In this assignment, you will make a system for reliable data transfer, so the receiver will get the same data as the sender has sent, even when some data got corrupted during transmission.[[1]](#footnote-1)

## Principles

### 4.2.1 Nibbles

The error correction of this assignment takes a *nibble* as its starting point. One data byte can be divided into two nibbles (see [https://en.wikipedia.org/wiki/Nibble)](https://en.wikipedia.org/wiki/Nibble):



d3

d2

d1

d0



d3

d2

d1

d0



High nibble

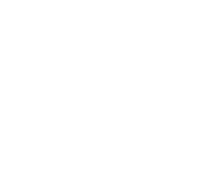
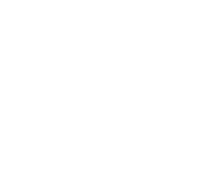


Low nibble

### 4.2.2 Parity bits

A nibble has 4 bits; let's call them data bits d0, d1, d2, d3. Draw 3 circles and put the data bits according the picture below. For each group, three parity bits (called p0, p1, p2) (see [https://en.wikipedia.org/wiki/Parity\_bit)](https://en.wikipedia.org/wiki/Parity_bit) are added in the picture.

The values of the parity bits are chosen in such a way that each circle always has an even parity (in other words: the sum of the bits in one circle must be even).



d0



d2



d1



d3



p0



p1



p2

Please check for yourself that the parities as given are correct.

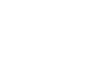
|  |  |
| --- | --- |
| 0    0    1    1    p0    p1    p2 | p0=1  p1=0  p2=0 |
|  |  |
| 1    0    1    1    p0    p1    p2 | p0=0  p1=1  p2=0 |
|  |  |
| 0    1    0    0    p0    p1    p2 | p0=1  p1=0  p2=1 |

## Software

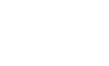
This paragraph gives the module design of the programs of this assignment. As you can read in the next paragraphs, there are 3 programs to be made:

* Part A: encode (at the sender's side)
* Part B: channel (represents the internet where errors might occur)
* Part C: decode (at the receiver's side)

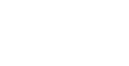
Their relation is the following:



**encode**



**decode**



**channel**



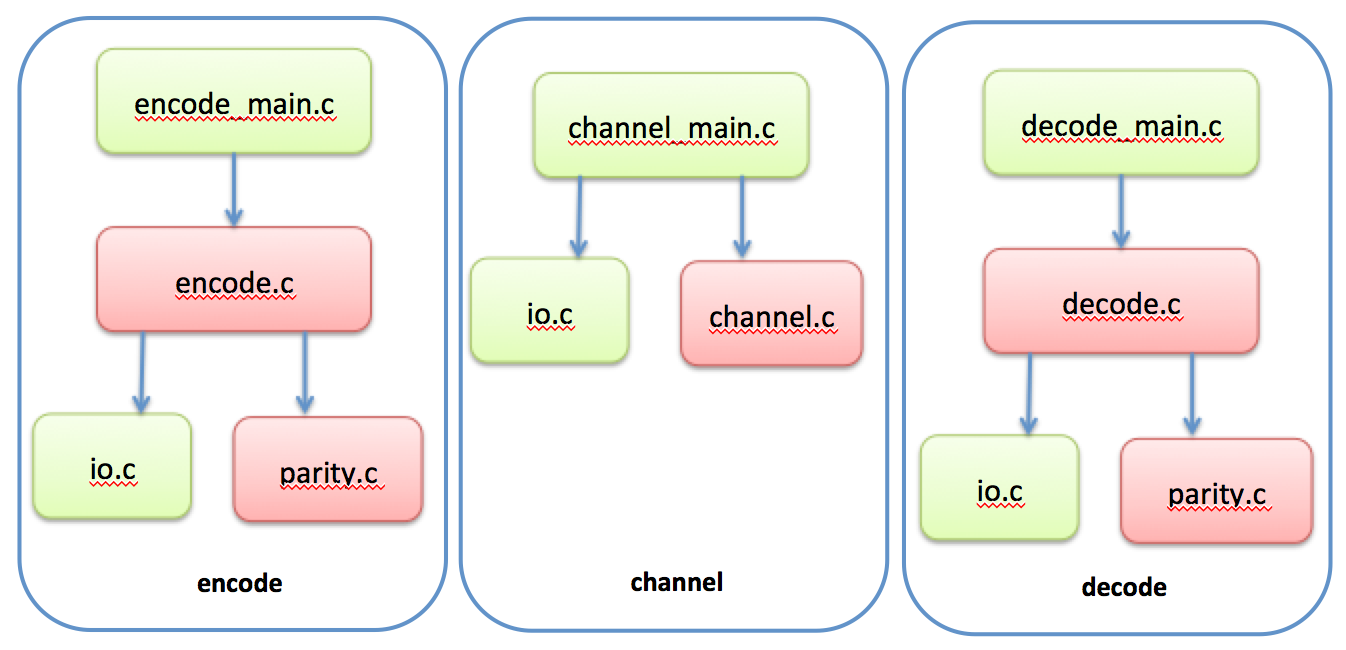
***ABC***



***ABC***

And of course with the goal that the text on the left ("ABC") is identical to the text on the right ("ABC")

### 3.3.1 Design

The programs are built with the following modules. As you can see the main programs share modules so there is definitely no code duplication! The files as marked in red still need to be implemented. 

|  |  |
| --- | --- |
| **Module** | **Description** |
| encode\_main | Contains main() for the encode program. |
| encode | Methods for encoding. |
| channel\_main | Contains main() for the channel program. |
| channel | Contains an operation for random bit flipping (to simulate a noisy internet). |
| decode\_main | Contains main() for the decode program. |
| decode | Methods for decoding. |
| parity | Contains all parity handling (calculating + adding the parity bits, removing parity bits + perhaps correcting data bits). |
| io | Contains all file handling (opening. closing, reading, writing). |

Tabel 4.1: Global description of the modules.

### 3.3.2 Files that need to be implemented

For this assignment you need to implement the files that are given in the table below.

|  |  |
| --- | --- |
| **To be implemented** | **Explanation** |
| encode.c [part-A],  channel.c [part-B],  decode.c [part-C] | The functions have already been defined in the .h files, so you don't need to modify the header files. You only have to implement the .c files. |
| parity.c/h [part-A+C] | These modules are empty and you have to define and implement the functions with their parameter list yourself. |
| encode\_test.c [part-A],  parity\_test.c [part-A+C],  channel\_test.c [part-B],  decode\_test.c [part-C] | Implementing unit tests for the corresponding modules. |

Tabel 4.2: Overview of files that need to be implemented. The [ ]-brackets show in which part of the assignment these files will be needed.

### 3.3.3 Build instructions

All files are already provided in the directory Adidas of directory StartPointForAdidas (but everal files are empty...). The Makefile is also already prepared, and you don't need to modify it.

|  |  |  |
| --- | --- | --- |
| **Assignment Part:** | **Build instructions:** | **Description:** |
| A / B / C | make | Builds the executable called ‘main’. See instructions in the following parts on how to use the ‘encode|channel|decode’ functionality. |
|  | make test | Builds the executable ‘main\_test’. |
| A: Encode | make encode\_test | Runs the **encode** unit-tests. |
| B: Channel | make channel\_test | Runs the **channel** unit-tests. |
| C: Decode | make decode\_test | Runs the **decode** unit-tests. |
| ultimate test | make run | Builds and runs main encode-channel-decode and compares the inputfile(makefile) with the output. These should be the same. |

Table 4.3: Build instructions for various modules and unit-tests.

## Part A: Encoding

Write a program that adds parity bits to the data, according to the description above. After building the main of the encoding program is started from a terminal as follows:

./build/main encode inputfile outputfile

inputfile is the original, existing file, outputfile is generated, with the parity bits.

The structure of the output file is:

byte 1: 4 data + 3 parity bits of *high* nibble of *first* input byte  
byte 2: 4 data + 3 parity bits of *low* nibble of *first* input byte

....  
…

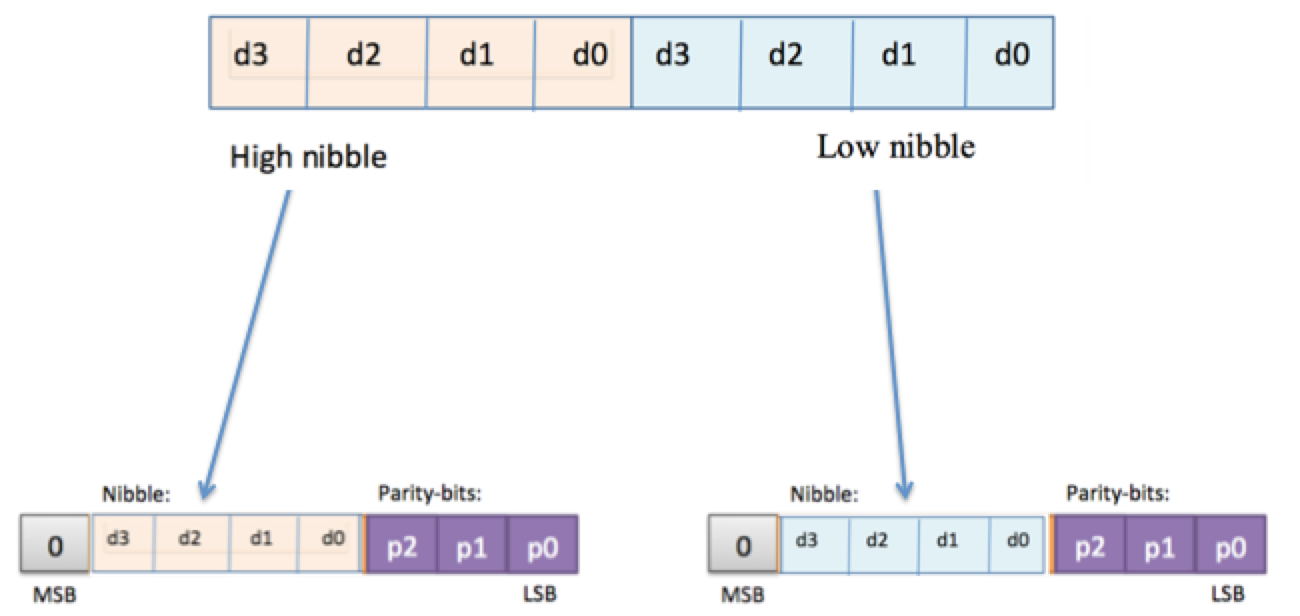
byte n-1: 4 data + 3 parity bits of *high* nibble of *last* input byte  
byte n: 4 data + 3 parity bits of *low* nibble of *last* input byte

In each byte, the 7 bits are stored as follows:



Example: if the first byte in the input file is character 'A', then byte 1 and byte 2 of the output file have the hexadecimal values: 0x25 0x0b (please check those values by hand!).

In figure below you can see the encoding of 1 byte into two bytes that each contain one of the nibbles and the corresponding parity bits.



Notes:

* All operations have to be done with proper bit-manipulations (masking, shifting etc.), so *not* with e.g. big switch-statements in which all 4 data x 3 parity possibilities are pre-coded. (but, of course, those pre-coded values may well be used for testing)

## Part B: Channel

Write a program that simulates errors during transmission. The program is started as follows:

./build/main channel inputfile outputfile

inputfile is an existing file (which might be the output file of the encoding).   
outputfile is based on the input file where some bits are flipped (i.e. changed from 0 to 1, or from 1 to zero) to simulate an interference during transmission.

The outputfile has the following characteristics:

* The odd bytes of the input file are unchanged (so the 1st, 3rd, 5th, .. byte is unchanged)
* For all other bytes: *one* random bit is flipped (for each bit there is a 1/8 probability that it is flipped)

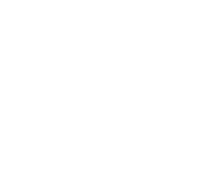
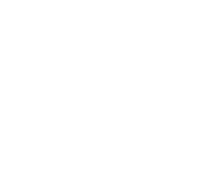
1. Indicate which parity bits (P0..P1) became inconsistent after one of the nibble bits (DO..D3) was flipped by the channel.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Flipped bit** | **Inconsistent  parity-bit(s)** | | D0 |  | | D1 |  | | D2 |  | | D3 |  | | d0    d2    d1    d3    p0    p1    p2 |

b) Implement the channel.c file. Use the rand()function to get random values. This module already calls srand() to make sure that you get different values (each time that you start the program).

## Part C: Decoding

Before you start the programming of this part, let's first do some exercises on paper to see how the error correction works.



d0



d2



d1



d3



p0



p1



p2

a) In each of the following situations, exactly *one* bit has been flipped. Discover

which bit has been flipped (so find out what the original values of the data bits d0..d3 have been).



1



0



1



0



1



1



0



0



0



1



0



0



1



0



0



1



1



1



0



1



1

b) Please fill in the last column to find out which bit (D0..D3 or P0..P2) was corrupted by noise:

|  |  |  |  |
| --- | --- | --- | --- |
| **P0-correct** | **P1-correct** | **P2-correct** | **Which bit was made incorrect?** |
| PASS | PASS | PASS | Possibly the MSB bit. |
| FAIL | PASS | PASS |  |
| PASS | FAIL | PASS |  |
| PASS | PASS | FAIL |  |
| FAIL | FAIL | PASS |  |
| PASS | FAIL | FAIL |  |
| FAIL | PASS | FAIL |  |
| FAIL | FAIL | FAIL |  |

c) Write a program to perform error correction (the decoding) on a file. The program is started as follows:

./build/main decode inputfile outputfile

inputfile is the file containing the received data, according to the output format as described in Part A (mode 1). outputfile contains the data after error corrections, identical to what the sender intended to transmit (so all errors are corrected, and all parity bits are removed).

# Submission

Only the files as mentioned in table 4.2 need to be submitted.

1. Please note: our implementation will only work when there is no more than *one* corrupted bit per byte (when the transmission channel gives *more* problems, then we have to select a more powerful system...) [↑](#footnote-ref-1)