

`cryptopals_c`

Generated by Doxygen 1.9.8

1 Cryptopals in C Code	1
1.1 Overview	1
1.2 Documentation	1
1.3 index.html	1
1.4 Requirements	1
1.5 Building & Running	2
1.6 Security Disclaimer	2
1.7 References	2
2 File Index	3
2.1 File List	3
3 File Documentation	5
3.1 set_1_challenge_1.c File Reference	5
3.1.1 Detailed Description	5
3.1.2 Function Documentation	5
3.1.2.1 main()	5
3.2 set_1_challenge_1.c	6
3.3 set_1_challenge_2.c File Reference	6
3.3.1 Detailed Description	7
3.3.2 Function Documentation	7
3.3.2.1 hex_digit_value()	7
3.3.2.2 hex_to_bytes()	7
3.3.2.3 main()	8
3.4 set_1_challenge_2.c	8
3.5 fixed_xor.h File Reference	9
3.5.1 Detailed Description	10
3.5.2 Enumeration Type Documentation	11
3.5.2.1 fixed_xor_status	11
3.5.3 Function Documentation	11
3.5.3.1 fixed_xor_buffers()	11
3.5.3.2 fixed_xor_status_string()	12
3.5.3.3 fixed_xor_stream()	12
3.6 fixed_xor.h	13
3.7 hex2b64.h File Reference	13
3.7.1 Detailed Description	14
3.7.2 Function Documentation	14
3.7.2.1 hex2b64_buffer()	14
3.7.2.2 hex2b64_stream()	15
3.8 hex2b64.h	16
3.9 score_english_hex.h File Reference	16
3.9.1 Detailed Description	17
3.9.2 Function Documentation	17

3.9.2.1 score_english_hex()	17
3.10 score_english_hex.h	18
3.11 fixed_xor.c File Reference	18
3.11.1 Detailed Description	19
3.11.2 Macro Definition Documentation	19
3.11.2.1 FIXED_XOR_CHUNK	19
3.11.3 Function Documentation	19
3.11.3.1 fixed_xor_buffers()	19
3.11.3.2 fixed_xor_grow()	19
3.11.3.3 fixed_xor_status_string()	20
3.11.3.4 fixed_xor_stream()	20
3.12 fixed_xor.c	21
3.13 hex2b64.c File Reference	23
3.13.1 Detailed Description	23
3.13.2 Function Documentation	23
3.13.2.1 encode_base64_block()	23
3.13.2.2 encode_base64_chars()	24
3.13.2.3 hex2b64_buffer()	25
3.13.2.4 hex2b64_stream()	25
3.13.2.5 hex_value()	26
3.13.3 Variable Documentation	27
3.13.3.1 b64_table	27
3.14 hex2b64.c	27
3.15 score_english_hex.c File Reference	29
3.15.1 Detailed Description	29
3.15.2 Function Documentation	29
3.15.2.1 hex_value()	29
3.15.2.2 score_english_hex()	30
3.15.3 Variable Documentation	31
3.15.3.1 english_freq	31
3.16 score_english_hex.c	31
3.17 README.md File Reference	32
3.18 fixed_xor_main.c File Reference	32
3.18.1 Detailed Description	33
3.18.2 Function Documentation	33
3.18.2.1 main()	33
3.19 fixed_xor_main.c	34
3.20 hex2b64_main.c File Reference	34
3.20.1 Detailed Description	34
3.20.2 Function Documentation	34
3.20.2.1 main()	34
3.21 hex2b64_main.c	35

3.22 score_english_hex_main.c File Reference	35
3.22.1 Detailed Description	35
3.22.2 Function Documentation	36
3.22.2.1 main()	36
3.22.2.2 score_to_percentage()	36
3.23 score_english_hex_main.c	36
Index	39

Chapter 1

Cryptopals in C Code

Solve the **Cryptopals Crypto Challenges** using **pure C**.

- This repository contains my implementations, notes, and explanations for each challenge set, focused on clarity, correctness, and learning modern applied cryptography from first principles.

Basically a quiet Saturday and I am in need of a challenge, so I am speedrunning the Cryptopals challenges in C code, just for fun.

1.1 Overview

The **Cryptopals Challenges** are a well-known set of practical cryptography exercises covering topics like:

- Encoding and decoding
- XOR ciphers
- Block ciphers (AES)
- Padding oracles
- CBC, CTR, stream attacks
- Diffie–Hellman, RSA, and more

This repo is my attempt to solve them **from scratch in C**, without relying on high-level crypto libraries unless the challenge's intent allows it.

The goal is to deeply understand how the primitives work internally.

1.2 Documentation

1.3 <[a href=".//docs/html/index.html" >index.html](#)

1.4 Requirements

- C99 or newer
 - Make
 - doxygen and pdflatex/texlive environment for docs
-

1.5 Building & Running

Common combinations of the below:

```
make clean  
make  
make test  
make docs
```

1.6 Security Disclaimer

- This code is for my own amusement and educational purposes only.
 - Do not use it in production, security-critical, or cryptographic applications.
-

1.7 References

[Cryptopals Challenges](#)

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

set_1_challenge_1.c	Cryptopals Set 1 Challenge 1 sample solution using <code>hex2b64_buffer()</code>	5
set_1_challenge_2.c	Cryptopals Set 1 Challenge 2: fixed XOR test harness	6
fixed_xor.h	Public interface for XORing two equally sized buffers	9
hex2b64.h	Public interface for converting hexadecimal data to Base64	13
score_english_hex.h	Estimate how closely hex-encoded data resembles English text	16
fixed_xor.c	Implementation of fixed-length buffer XOR helpers	18
hex2b64.c	Implementation of hex-to-Base64 conversion helpers	23
score_english_hex.c	Implementation of English scoring for hex-encoded strings	29
fixed_xor_main.c	Command-line wrapper for XORing two equal-length buffers	32
hex2b64_main.c	Command-line tool that converts hex input to Base64	34
score_english_hex_main.c	CLI utility that scores hex data for English-likeness	35

Chapter 3

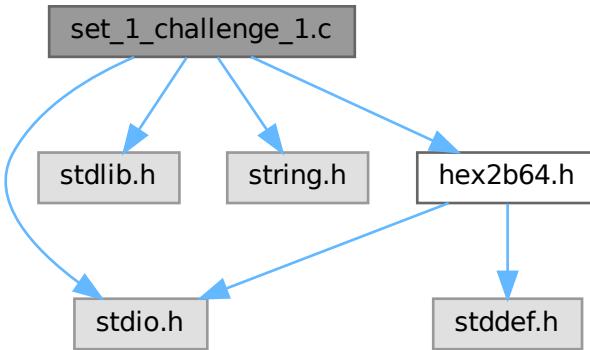
File Documentation

3.1 set_1_challenge_1.c File Reference

Cryptopals Set 1 Challenge 1 sample solution using [hex2b64_buffer\(\)](#).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hex2b64.h"
```

Include dependency graph for set_1_challenge_1.c:



Functions

- int [main](#) (void)

Convert the fixed challenge hex string and compare to expected Base64.

3.1.1 Detailed Description

Cryptopals Set 1 Challenge 1 sample solution using [hex2b64_buffer\(\)](#).

Definition in file [set_1_challenge_1.c](#).

3.1.2 Function Documentation

3.1.2.1 main()

```
int main (
```

```
    void  )
```

Convert the fixed challenge hex string and compare to expected Base64.

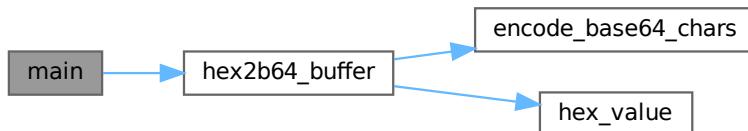
Returns

`EXIT_SUCCESS` when the conversion matches the known answer.

Definition at line 17 of file [set_1_challenge_1.c](#).

References [hex2b64_buffer\(\)](#).

Here is the call graph for this function:



3.2 set_1_challenge_1.c

[Go to the documentation of this file.](#)

```

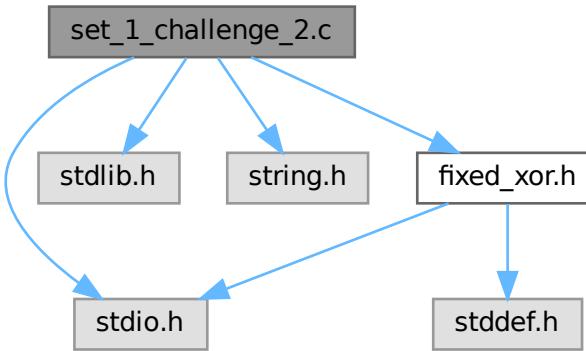
00001
00006 #include <stdio.h>
00007 #include <stdlib.h>
00008 #include <string.h>
00009
00010 #include "hex2b64.h"
00011
00017 int main(void) {
00018     const char *hex_input =
00019         "49276d206b696c6c696e6720796f757220627261696e206c696b65206120706f69736f6e6f7573206d757368726f6f6d";
00020     const char *expected_b64 = "SSdtIGtpbGxpibmcgeW9lciBicmFpbisWt1IGEgcG9pc29ub3VzIG1lc2hyb29t\n";
00021
00022     unsigned char buffer[256];
00023     size_t out_len = 0;
00024     int rc = hex2b64_buffer((const unsigned char *)hex_input,
00025                             strlen(hex_input),
00026                             buffer,
00027                             sizeof(buffer),
00028                             &out_len);
00029     if (rc != 0) {
00030         fprintf(stderr, "hex2b64_buffer failed\n");
00031         return EXIT_FAILURE;
00032     }
00033     buffer[out_len] = '\0';
00034
00035     if (strcmp((const char *)buffer, expected_b64) == 0) {
00036         fprintf(stdout, "PASS: %s == %s\n", hex_input, expected_b64);
00037         return EXIT_SUCCESS;
00038     }
00039
00040     fprintf(stdout, "FAIL: %s != %s\n", hex_input, expected_b64);
00041     return EXIT_FAILURE;
00042 }
```

3.3 set_1_challenge_2.c File Reference

Cryptopals Set 1 Challenge 2: fixed XOR test harness.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "fixed_xor.h"
```

Include dependency graph for set_1_challenge_2.c:



Functions

- static int [hex_digit_value](#) (char c)
- static int [hex_to_bytes](#) (const char *hex, unsigned char *out, size_t out_cap)
- int [main](#) (void)

3.3.1 Detailed Description

Cryptopals Set 1 Challenge 2: fixed XOR test harness.

Definition in file [set_1_challenge_2.c](#).

3.3.2 Function Documentation

3.3.2.1 hex_digit_value()

```
static int hex_digit_value (
    char c ) [static]
```

Definition at line 12 of file [set_1_challenge_2.c](#).

Here is the caller graph for this function:



3.3.2.2 hex_to_bytes()

```
static int hex_to_bytes (
    const char * hex,
    unsigned char * out,
    size_t out_cap ) [static]
```

Definition at line 19 of file [set_1_challenge_2.c](#).

References [hex_digit_value\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



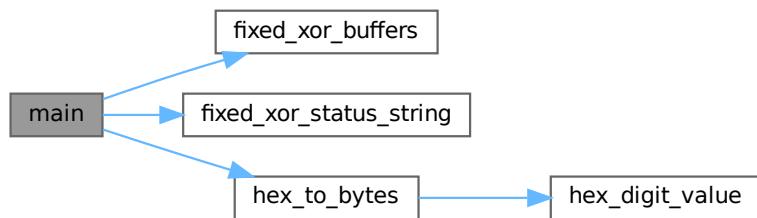
3.3.2.3 main()

```
int main (
    void )
```

Definition at line 40 of file [set_1_challenge_2.c](#).

References [fixed_xor_buffers\(\)](#), [FIXED_XOR_OK](#), [fixed_xor_status_string\(\)](#), and [hex_to_bytes\(\)](#).

Here is the call graph for this function:



3.4 set_1_challenge_2.c

[Go to the documentation of this file.](#)

```
00001
00006 #include <stdio.h>
00007 #include <stdlib.h>
00008 #include <string.h>
00009
00010 #include "fixed_xor.h"
00011
00012 static int hex_digit_value(char c) {
00013     if (c >= '0' && c <= '9') return c - '0';
  
```

```

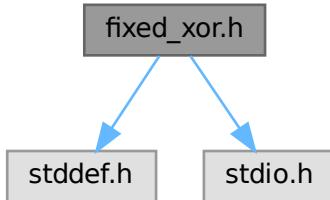
00014     if (c >= 'a' && c <= 'f') return 10 + (c - 'a');
00015     if (c >= 'A' && c <= 'F') return 10 + (c - 'A');
00016     return -1;
00017 }
00018
00019 static int hex_to_bytes(const char *hex, unsigned char *out, size_t out_cap) {
00020     size_t hex_len = strlen(hex);
00021     if (hex_len % 2 != 0) {
00022         return -1;
00023     }
00024     size_t byte_len = hex_len / 2;
00025     if (byte_len > out_cap) {
00026         return -1;
00027     }
00028
00029     for (size_t i = 0; i < byte_len; ++i) {
00030         int hi = hex_digit_value(hex[2 * i]);
00031         int lo = hex_digit_value(hex[2 * i + 1]);
00032         if (hi < 0 || lo < 0) {
00033             return -1;
00034         }
00035         out[i] = (unsigned char)((hi << 4) | lo);
00036     }
00037     return (int)byte_len;
00038 }
00039
00040 int main(void) {
00041     const char *lhs_hex = "1c0111001f010100061a024b53535009181c";
00042     const char *rhs_hex = "686974207468652062756c6c277320657965";
00043     const char *expected_hex = "746865206b696420646f6e277420706c6179";
00044
00045     unsigned char lhs[32];
00046     unsigned char rhs[32];
00047     unsigned char expected[32];
00048     unsigned char result[32];
00049
00050     int lhs_len = hex_to_bytes(lhs_hex, lhs, sizeof(lhs));
00051     int rhs_len = hex_to_bytes(rhs_hex, rhs, sizeof(rhs));
00052     int expected_len = hex_to_bytes(expected_hex, expected, sizeof(expected));
00053
00054     if (lhs_len < 0 || rhs_len < 0 || expected_len < 0 ||
00055         lhs_len != rhs_len || lhs_len != expected_len) {
00056         fprintf(stderr, "failed to decode hex inputs\n");
00057         return EXIT_FAILURE;
00058     }
00059
00060     fixed_xor_status status =
00061         fixed_xor_buffers(lhs, rhs, result, (size_t)lhs_len);
00062     if (status != FIXED_XOR_OK) {
00063         fprintf(stderr, "fixed_xor_buffers failed: %s\n",
00064                 fixed_xor_status_string(status));
00065         return EXIT_FAILURE;
00066     }
00067
00068     if (memcmp(result, expected, (size_t)expected_len) == 0) {
00069         printf("PASS: %s XOR %s == %s\n", lhs_hex, rhs_hex, expected_hex);
00070         return EXIT_SUCCESS;
00071     }
00072
00073     printf("FAIL: result does not match expected output\n");
00074     return EXIT_FAILURE;
00075 }
```

3.5 fixed_xor.h File Reference

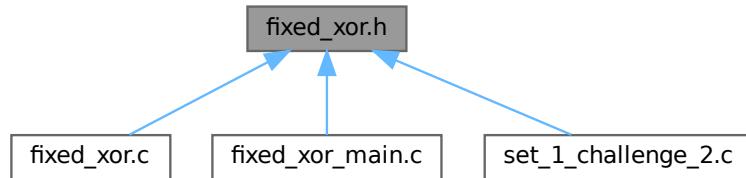
Public interface for XORing two equally sized buffers.

```
#include <stddef.h>
#include <stdio.h>
```

Include dependency graph for fixed_xor.h:



This graph shows which files directly or indirectly include this file:



Enumerations

- enum `fixed_xor_status` {
 `FIXED_XOR_OK` = 0 , `FIXED_XOR_ERR_ARGS` = -1 , `FIXED_XOR_ERR_IO` = -2 , `FIXED_XOR_ERR_ODD_INPUT` = -3 ,
 `FIXED_XOR_ERR_OOM` = -4 }

Status codes describing the outcome of fixed XOR operations.

Functions

- `fixed_xor_status fixed_xor_buffers` (const unsigned char *lhs, const unsigned char *rhs, unsigned char *out, size_t len)

XOR two buffers of equal length into an output buffer.

- `fixed_xor_status fixed_xor_stream` (FILE *in, FILE *out)

XOR two half-length buffers read sequentially from a stream.

- const char * `fixed_xor_status_string` (`fixed_xor_status` status)

Convert a `fixed_xor_status` value into a human-readable string.

3.5.1 Detailed Description

Public interface for XORing two equally sized buffers.

Definition in file [fixed_xor.h](#).

3.5.2 Enumeration Type Documentation

3.5.2.1 fixed_xor_status

```
enum fixed_xor_status
```

Status codes describing the outcome of fixed XOR operations.

Enumerator

FIXED_XOR_OK	Operation completed successfully.
FIXED_XOR_ERR_ARGS	Invalid arguments were supplied.
FIXED_XOR_ERR_IO	I/O failure while reading or writing.
FIXED_XOR_ERR_ODD_INPUT	Stream input contained an odd byte count.
FIXED_XOR_ERR_OOM	Memory allocation failed.

Definition at line 15 of file [fixed_xor.h](#).

3.5.3 Function Documentation

3.5.3.1 fixed_xor_buffers()

```
fixed_xor_status fixed_xor_buffers (
    const unsigned char * lhs,
    const unsigned char * rhs,
    unsigned char * out,
    size_t len )
```

XOR two buffers of equal length into an output buffer.

Parameters

<i>lhs</i>	Pointer to the left-hand buffer.
<i>rhs</i>	Pointer to the right-hand buffer.
<i>out</i>	Destination buffer that receives <i>lhs</i> \wedge <i>rhs</i> .
<i>len</i>	Number of bytes to process.

Returns

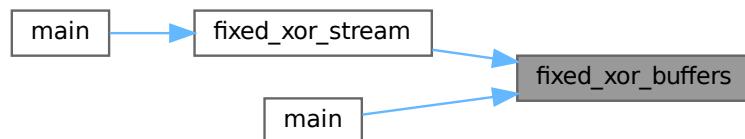
FIXED_XOR_OK on success or an error status on failure.

XOR two buffers of equal length into an output buffer.

Definition at line 18 of file [fixed_xor.c](#).

References [FIXED_XOR_ERR_ARGS](#), and [FIXED_XOR_OK](#).

Here is the caller graph for this function:



3.5.3.2 fixed_xor_status_string()

```
const char * fixed_xor_status_string (
    fixed_xor_status status )
```

Convert a fixed_xor_status value into a human-readable string.

Parameters

<code>status</code>	Status code to describe.
---------------------	--------------------------

Returns

Pointer to a static string literal.

Convert a fixed_xor_status value into a human-readable string.

Definition at line 125 of file [fixed_xor.c](#).

References [FIXED_XOR_ERR_ARGS](#), [FIXED_XOR_ERR_IO](#), [FIXED_XOR_ERR_ODD_INPUT](#), [FIXED_XOR_ERR_OOM](#), and [FIXED_XOR_OK](#).

Here is the caller graph for this function:



3.5.3.3 fixed_xor_stream()

```
fixed_xor_status fixed_xor_stream (
    FILE * in,
    FILE * out )
```

XOR two half-length buffers read sequentially from a stream.

The input stream is expected to contain two equally sized buffers back to back. The result is written to `out`.

Parameters

<code>in</code>	Stream containing the concatenated buffers.
<code>out</code>	Stream that receives XOR output.

Returns

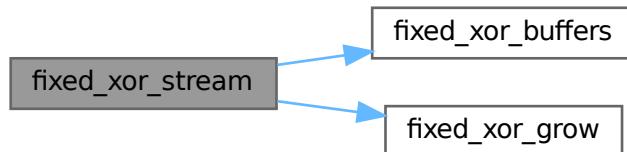
`FIXED_XOR_OK` on success or an error status on failure.

XOR two half-length buffers read sequentially from a stream.

Definition at line 68 of file [fixed_xor.c](#).

References `fixed_xor_buffers()`, `FIXED_XOR_CHUNK`, `FIXED_XOR_ERR_ARGS`, `FIXED_XOR_ERR_IO`, `FIXED_XOR_ERR_ODD_INPUT`, `FIXED_XOR_ERR_OOM`, `fixed_xor_grow()`, and `FIXED_XOR_OK`.

Here is the call graph for this function:



Here is the caller graph for this function:



3.6 fixed_xor.h

[Go to the documentation of this file.](#)

```

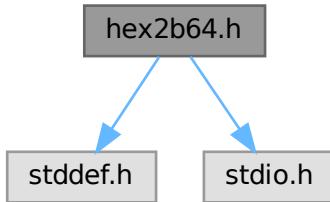
00001 #ifndef FIXED_XOR_H
00002 #define FIXED_XOR_H
00003
00009 #include <stddef.h>
00010 #include <stdio.h>
00011
00015 typedef enum {
00016     FIXED_XOR_OK = 0,
00017     FIXED_XOR_ERR_ARGS = -1,
00018     FIXED_XOR_ERR_IO = -2,
00019     FIXED_XOR_ERR_ODD_INPUT = -3,
00020     FIXED_XOR_ERR_OOM = -4
00021 } fixed_xor_status;
00022
00032 fixed_xor_status fixed_xor_buffers(const unsigned char *lhs,
00033                                     const unsigned char *rhs,
00034                                     unsigned char *out,
00035                                     size_t len);
00036
00047 fixed_xor_status fixed_xor_stream(FILE *in, FILE *out);
00048
00055 const char *fixed_xor_status_string(fixed_xor_status status);
00056
00057 #endif /* FIXED_XOR_H */

```

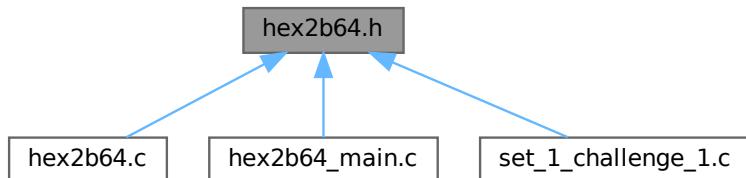
3.7 hex2b64.h File Reference

Public interface for converting hexadecimal data to Base64.

```
#include <stddef.h>
#include <stdio.h>
Include dependency graph for hex2b64.h:
```



This graph shows which files directly or indirectly include this file:



Functions

- int `hex2b64_stream` (FILE *in, FILE *out)
Convert hexadecimal text read from a stream into Base64.
- int `hex2b64_buffer` (const unsigned char *hex, size_t hex_len, unsigned char *out, size_t out_cap, size_t *out_len)
Convert a memory buffer of hexadecimal characters into Base64.

3.7.1 Detailed Description

Public interface for converting hexadecimal data to Base64.

Definition in file `hex2b64.h`.

3.7.2 Function Documentation

3.7.2.1 `hex2b64_buffer()`

```
int hex2b64_buffer (
    const unsigned char * hex,
    size_t hex_len,
    unsigned char * out,
    size_t out_cap,
    size_t * out_len )
```

Convert a memory buffer of hexadecimal characters into Base64.

The input buffer may contain ASCII whitespace, which is skipped. The output buffer receives the Base64 encoding followed by a newline. The caller must supply sufficient capacity (including room for the newline). When `out_len` is non-NULL it receives the number of bytes written.

Parameters

<code>hex</code>	Pointer to the hex buffer (may be NULL when <code>hex_len</code> is 0).
<code>hex_len</code>	Number of bytes in <code>hex</code> .
<code>out</code>	Destination buffer for Base64 characters.
<code>out_cap</code>	Capacity of <code>out</code> in bytes.
<code>out_len</code>	Optional pointer that receives the bytes produced.

Returns

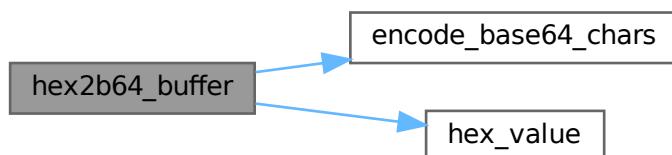
0 on success, non-zero on invalid hex, odd digit count, or overflow.

Convert a memory buffer of hexadecimal characters into Base64.

Definition at line 123 of file [hex2b64.c](#).

References [encode_base64_chars\(\)](#), and [hex_value\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



3.7.2.2 hex2b64_stream()

```
int hex2b64_stream (
    FILE * in,
    FILE * out )
```

Convert hexadecimal text read from a stream into Base64.

Whitespace characters in the input stream are ignored. The output stream receives the Base64 encoding plus a trailing newline. Errors are reported via the return code and (for malformed hex) an explanatory message on stderr.

Parameters

<i>in</i>	Input stream providing ASCII hex characters.
<i>out</i>	Output stream that receives Base64 data.

Returns

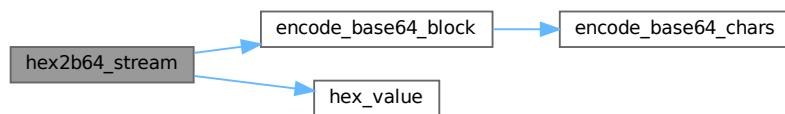
0 on success, non-zero on invalid input or I/O failure.

Convert hexadecimal text read from a stream into Base64.

Definition at line 70 of file [hex2b64.c](#).

References [encode_base64_block\(\)](#), and [hex_value\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



3.8 hex2b64.h

[Go to the documentation of this file.](#)

```

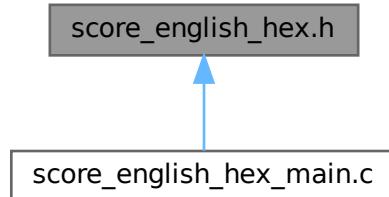
00001 #ifndef HEX2B64_H
00002 #define HEX2B64_H
00003
00009 #include <stddef.h>
00010 #include <stdio.h>
00011
00023 int hex2b64_stream(FILE *in, FILE *out);
00024
00040 int hex2b64_buffer(const unsigned char *hex,
00041                      size_t hex_len,
00042                      unsigned char *out,
00043                      size_t out_cap,
00044                      size_t *out_len);
00045
00046 #endif /* HEX2B64_H */

```

3.9 score_english_hex.h File Reference

Estimate how closely hex-encoded data resembles English text.

This graph shows which files directly or indirectly include this file:



Functions

- double [score_english_hex](#) (const char *hex)
Score a hex string based on English letter frequency heuristics.

3.9.1 Detailed Description

Estimate how closely hex-encoded data resembles English text.
Definition in file [score_english_hex.h](#).

3.9.2 Function Documentation

3.9.2.1 [score_english_hex\(\)](#)

```
double score_english_hex (
    const char * hex )
```

Score a hex string based on English letter frequency heuristics.

Parameters

hex	Null-terminated ASCII hex string.
-----	-----------------------------------

Returns

Higher values indicate closer resemblance to English text.

Score a hex string based on English letter frequency heuristics.
Definition at line 59 of file [score_english_hex.c](#).
References [english_freq](#), and [hex_value\(\)](#).
Here is the call graph for this function:



Here is the caller graph for this function:



3.10 score_english_hex.h

[Go to the documentation of this file.](#)

```

00001 #ifndef SCORE_ENGLISH_HEX_H
00002 #define SCORE_ENGLISH_HEX_H
00003
00015 double score_english_hex(const char *hex);
00016
00017 #endif /* SCORE_ENGLISH_HEX_H */
  
```

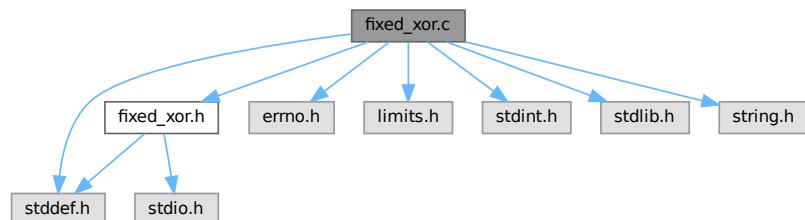
3.11 fixed_xor.c File Reference

Implementation of fixed-length buffer XOR helpers.

```

#include "fixed_xor.h"
#include <errno.h>
#include <limits.h>
#include <stddef.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
  
```

Include dependency graph for fixed_xor.c:



Macros

- [#define FIXED_XOR_CHUNK 4096](#)

Functions

- [fixed_xor_status fixed_xor_buffers](#) (const unsigned char *lhs, const unsigned char *rhs, unsigned char *out, size_t len)

Implementation of [fixed_xor_buffers\(\)](#).
- static [fixed_xor_status fixed_xor_grow](#) (unsigned char **buffer, size_t *capacity, size_t required)

Ensure that buffer has capacity for required bytes.

- `fixed_xor_status fixed_xor_stream (FILE *in, FILE *out)`
Implementation of `fixed_xor_stream()`.
- `const char * fixed_xor_status_string (fixed_xor_status status)`
Implementation of `fixed_xor_status_string()`.

3.11.1 Detailed Description

Implementation of fixed-length buffer XOR helpers.

Definition in file [fixed_xor.c](#).

3.11.2 Macro Definition Documentation

3.11.2.1 FIXED_XOR_CHUNK

```
#define FIXED_XOR_CHUNK 4096
```

Definition at line 15 of file [fixed_xor.c](#).

3.11.3 Function Documentation

3.11.3.1 fixed_xor_buffers()

```
fixed_xor_status fixed_xor_buffers (
    const unsigned char * lhs,
    const unsigned char * rhs,
    unsigned char * out,
    size_t len )
```

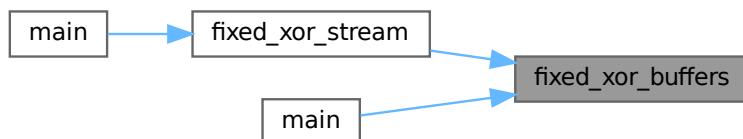
Implementation of [fixed_xor_buffers\(\)](#).

XOR two buffers of equal length into an output buffer.

Definition at line 18 of file [fixed_xor.c](#).

References [FIXED_XOR_ERR_ARGS](#), and [FIXED_XOR_OK](#).

Here is the caller graph for this function:



3.11.3.2 fixed_xor_grow()

```
static fixed_xor_status fixed_xor_grow (
    unsigned char ** buffer,
    size_t * capacity,
    size_t required ) [static]
```

Ensure that `buffer` has capacity for `required` bytes.

Parameters

<code>buffer</code>	Pointer to heap storage pointer to grow.
<code>capacity</code>	Current capacity in bytes (updated on success).
<code>required</code>	Target capacity required by the caller.

Returns

`FIXED_XOR_OK` on success or `FIXED_XOR_ERR_OOM` on allocation failure.

Definition at line 42 of file [fixed_xor.c](#).

References [FIXED_XOR_ERR_OOM](#), and [FIXED_XOR_OK](#).

Here is the caller graph for this function:

**3.11.3.3 fixed_xor_status_string()**

```
const char * fixed_xor_status_string (
    fixed_xor_status status )
```

Implementation of [fixed_xor_status_string\(\)](#).

Convert a `fixed_xor_status` value into a human-readable string.

Definition at line 125 of file [fixed_xor.c](#).

References [FIXED_XOR_ERR_ARGS](#), [FIXED_XOR_ERR_IO](#), [FIXED_XOR_ERR_ODD_INPUT](#), [FIXED_XOR_ERR_OOM](#), and [FIXED_XOR_OK](#).

Here is the caller graph for this function:

**3.11.3.4 fixed_xor_stream()**

```
fixed_xor_status fixed_xor_stream (
    FILE * in,
    FILE * out )
```

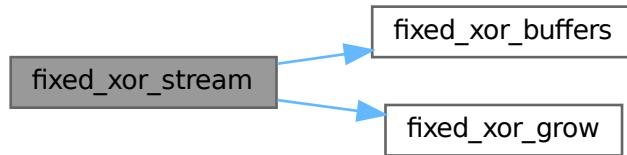
Implementation of [fixed_xor_stream\(\)](#).

XOR two half-length buffers read sequentially from a stream.

Definition at line 68 of file [fixed_xor.c](#).

References [fixed_xor_buffers\(\)](#), [FIXED_XOR_CHUNK](#), [FIXED_XOR_ERR_ARGS](#), [FIXED_XOR_ERR_IO](#), [FIXED_XOR_ERR_ODD_INPUT](#), [FIXED_XOR_ERR_OOM](#), [fixed_xor_grow\(\)](#), and [FIXED_XOR_OK](#).

Here is the call graph for this function:



Here is the caller graph for this function:



3.12 fixed_xor.c

[Go to the documentation of this file.](#)

```

00001
00006 #include "fixed_xor.h"
00007
00008 #include <errno.h>
00009 #include <limits.h>
00010 #include <stddef.h>
00011 #include <stdint.h>
00012 #include <stdlib.h>
00013 #include <string.h>
00014
00015 #define FIXED_XOR_CHUNK 4096
00016
00018 fixed_xor_status fixed_xor_buffers(const unsigned char *lhs,
00019                                     const unsigned char *rhs,
00020                                     unsigned char *out,
00021                                     size_t len) {
00022     if (!lhs || !rhs || !out) {
00023         errno = EINVAL;
00024         return FIXED_XOR_ERR_ARGS;
00025     }
00026
00027     for (size_t i = 0; i < len; ++i) {
00028         out[i] = lhs[i] ^ rhs[i];
00029     }
00030
00031     return FIXED_XOR_OK;
00032 }
00033
00042 static fixed_xor_status fixed_xor_grow(unsigned char **buffer,
00043                                         size_t *capacity,
00044                                         size_t required) {
00045     size_t new_capacity = *capacity;
00046     while (required > new_capacity) {
00047         if (new_capacity > (SIZE_MAX / 2)) {
00048             return FIXED_XOR_ERR_OOM;
00049         }
00050         new_capacity *= 2;
00051     }
00052
00053     if (new_capacity == *capacity) {
00054         return FIXED_XOR_OK;

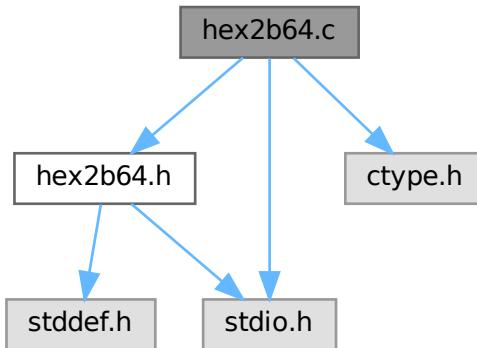
```

```
00055     }
00056
00057     unsigned char *new_data = realloc(*buffer, new_capacity);
00058     if (!new_data) {
00059         return FIXED_XOR_ERR_OOM;
00060     }
00061
00062     *buffer = new_data;
00063     *capacity = new_capacity;
00064     return FIXED_XOR_OK;
00065 }
00066
00067 fixed_xor_status fixed_xor_stream(FILE *in, FILE *out) {
00068     if (!in || !out) {
00069         errno = EINVAL;
00070         return FIXED_XOR_ERR_ARGS;
00071     }
00072
00073     size_t capacity = FIXED_XOR_CHUNK;
00074     unsigned char *data = malloc(capacity);
00075     if (!data) {
00076         return FIXED_XOR_ERR_OOM;
00077     }
00078
00079     size_t length = 0;
00080     unsigned char chunk[FIXED_XOR_CHUNK];
00081     size_t nread;
00082
00083     while ((nread = fread(chunk, 1, sizeof(chunk), in)) > 0) {
00084         if (length + nread > capacity) {
00085             fixed_xor_status grow_status =
00086                 fixed_xor_grow(&data, &capacity, length + nread);
00087             if (grow_status != FIXED_XOR_OK) {
00088                 free(data);
00089                 return grow_status;
00090             }
00091         }
00092         memcpy(data + length, chunk, nread);
00093         length += nread;
00094     }
00095
00096     if (ferror(in)) {
00097         free(data);
00098         return FIXED_XOR_ERR_IO;
00099     }
00100 }
00101
00102     if (length % 2 != 0) {
00103         free(data);
00104         return FIXED_XOR_ERR_ODD_INPUT;
00105     }
00106
00107     size_t half = length / 2;
00108     fixed_xor_status status = fixed_xor_buffers(data, data + half, data, half);
00109     if (status != FIXED_XOR_OK) {
00110         free(data);
00111         return status;
00112     }
00113
00114     size_t nwritten = fwrite(data, 1, half, out);
00115     if (nwritten != half) {
00116         free(data);
00117         return FIXED_XOR_ERR_IO;
00118     }
00119
00120     free(data);
00121     return FIXED_XOR_OK;
00122 }
00123
00124 const char *fixed_xor_status_string(fixed_xor_status status) {
00125     switch (status) {
00126         case FIXED_XOR_OK:
00127             return "success";
00128         case FIXED_XOR_ERR_ARGS:
00129             return "invalid arguments";
00130         case FIXED_XOR_ERR_IO:
00131             return "I/O failure";
00132         case FIXED_XOR_ERR_ODD_INPUT:
00133             return "input length must be even (two equal buffers)";
00134         case FIXED_XOR_ERR_OOM:
00135             return "out of memory";
00136         default:
00137             return "unknown error";
00138     }
00139 }
00140 }
```

3.13 hex2b64.c File Reference

Implementation of hex-to-Base64 conversion helpers.

```
#include "hex2b64.h"
#include <ctype.h>
#include <stdio.h>
Include dependency graph for hex2b64.c:
```



Functions

- static int [hex_value](#) (int c)
Convert a single hexadecimal character into its integer value.
- static void [encode_base64_chars](#) (const unsigned char *in, size_t len, char encoded[4])
Produce four Base64 characters from up to three bytes.
- static void [encode_base64_block](#) (const unsigned char *in, size_t len, FILE *out)
Write a Base64 block derived from up to three input bytes.
- int [hex2b64_stream](#) (FILE *in, FILE *out)
Implementation of [hex2b64_stream\(\)](#).
- int [hex2b64_buffer](#) (const unsigned char *hex, size_t hex_len, unsigned char *out, size_t out_cap, size_t *out_len)
Implementation of [hex2b64_buffer\(\)](#).

Variables

- static const char [b64_table](#) []

3.13.1 Detailed Description

Implementation of hex-to-Base64 conversion helpers.

Definition in file [hex2b64.c](#).

3.13.2 Function Documentation

3.13.2.1 [encode_base64_block\(\)](#)

```
static void encode_base64_block (
    const unsigned char * in,
```

```
size_t len,
FILE * out ) [static]
```

Write a Base64 block derived from up to three input bytes.

Parameters

<i>in</i>	Source bytes.
<i>len</i>	Number of source bytes.
<i>out</i>	Output stream that receives four Base64 characters.

Definition at line 59 of file [hex2b64.c](#).

References [encode_base64_chars\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



3.13.2.2 encode_base64_chars()

```
static void encode_base64_chars (
    const unsigned char * in,
    size_t len,
    char encoded[4] ) [static]
```

Produce four Base64 characters from up to three bytes.

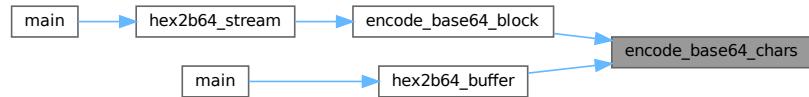
Parameters

<i>in</i>	Input bytes (padded implicitly with zeros).
<i>len</i>	Number of valid bytes (1-3).
<i>encoded</i>	Destination for the four encoded characters.

Definition at line 36 of file [hex2b64.c](#).

References [b64_table](#).

Here is the caller graph for this function:



3.13.2.3 hex2b64_buffer()

```
int hex2b64_buffer (
    const unsigned char * hex,
    size_t hex_len,
    unsigned char * out,
    size_t out_cap,
    size_t * out_len )
```

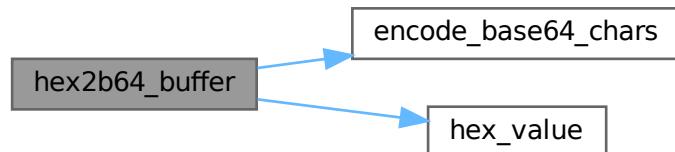
Implementation of [hex2b64_buffer\(\)](#).

Convert a memory buffer of hexadecimal characters into Base64.

Definition at line 123 of file [hex2b64.c](#).

References [encode_base64_chars\(\)](#), and [hex_value\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



3.13.2.4 hex2b64_stream()

```
int hex2b64_stream (
    FILE * in,
    FILE * out )
```

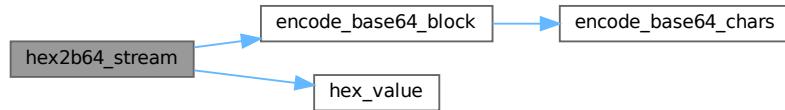
Implementation of [hex2b64_stream\(\)](#).

Convert hexadecimal text read from a stream into Base64.

Definition at line 70 of file [hex2b64.c](#).

References [encode_base64_block\(\)](#), and [hex_value\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



3.13.2.5 `hex_value()`

```
static int hex_value (
    int c ) [static]
```

Convert a single hexadecimal character into its integer value.

Parameters

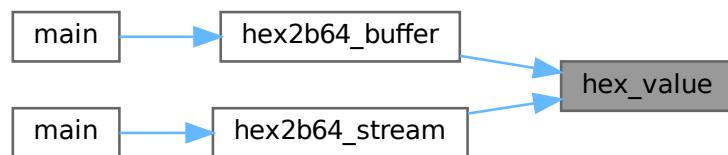
<code>c</code>	Character to convert.
----------------	-----------------------

Returns

0-15 for valid hex digits, or -1 when `c` is not hexadecimal.

Definition at line 22 of file [hex2b64.c](#).

Here is the caller graph for this function:



3.13.3 Variable Documentation

3.13.3.1 b64_table

```
const char b64_table[] [static]
```

Initial value:

```
=
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
"abcdefghijklmnopqrstuvwxyz"
"0123456789+/"
```

Definition at line 11 of file [hex2b64.c](#).

3.14 hex2b64.c

[Go to the documentation of this file.](#)

```
00001
00006 #include "hex2b64.h"
00007
00008 #include <ctype.h>
00009 #include <stdio.h>
00010
00011 static const char b64_table[] =
00012     "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
00013     "abcdefghijklmnopqrstuvwxyz"
00014     "0123456789+/";
00015
00022 static int hex_value(int c) {
00023     if (c >= '0' && c <= '9') return c - '0';
00024     if (c >= 'a' && c <= 'f') return 10 + (c - 'a');
00025     if (c >= 'A' && c <= 'F') return 10 + (c - 'A');
00026     return -1;
00027 }
00028
00036 static void encode_base64_chars(const unsigned char *in,
00037                                     size_t len,
00038                                     char encoded[4]) {
00039     unsigned int triple = 0;
00040
00041     /* Pack bytes into 24 bits: [b0 b1 b2] -> 0x00b0b1b2 */
00042     triple |= (unsigned int)in[0] << 16;
00043     if (len > 1) triple |= (unsigned int)in[1] << 8;
00044     if (len > 2) triple |= (unsigned int)in[2];
00045
00046     encoded[0] = b64_table[(triple >> 18) & 0x3F];
00047     encoded[1] = b64_table[(triple >> 12) & 0x3F];
00048     encoded[2] = (len > 1) ? b64_table[(triple >> 6) & 0x3F] : '=';
00049     encoded[3] = (len > 2) ? b64_table[triple & 0x3F] : '=';
00050 }
00051
00059 static void encode_base64_block(const unsigned char *in, size_t len, FILE *out) {
00060     char encoded[4];
00061     encode_base64_chars(in, len, encoded);
00062
00063     fputc(encoded[0], out);
00064     fputc(encoded[1], out);
00065     fputc(encoded[2], out);
00066     fputc(encoded[3], out);
00067 }
00068
00070 int hex2b64_stream(FILE *in, FILE *out) {
00071     int ch;
00072     int high_nibble = -1;           /* -1 means "no pending half-byte" */
00073     unsigned char buffer[3];        /* group bytes into blocks of 3 for Base64 */
00074     size_t buf_len = 0;
00075
00076     while ((ch = fgetc(in)) != EOF) {
00077         if (isspace((unsigned char)ch)) {
00078             /* Ignore whitespace entirely */
00079             continue;
00080         }
00081
00082         int v = hex_value(ch);
00083         if (v < 0) {
00084             fprintf(stderr, "Error: invalid hex character '%c'\n", ch);
00085             return 1;
00086         }
00087
00088         if (high_nibble < 0) {
00089             /* First half of the byte */
00090             high_nibble = v;
00091         } else {
00092             /* Second half: form full byte */
```

```

00093     unsigned char byte = (unsigned char)((high_nibble << 4) | v);
00094     high_nibble = -1;
00095
00096     buffer[buf_len++] = byte;
00097
00098     if (buf_len == 3) {
00099         encode_base64_block(buffer, buf_len, out);
00100         buf_len = 0;
00101     }
00102 }
00103 }
00104
00105 if (high_nibble >= 0) {
00106     /* Odd number of hex digits: cannot form a full byte */
00107     fprintf(stderr, "Error: odd number of hex digits (incomplete byte)\n");
00108     return 1;
00109 }
00110
00111 /* Flush remaining bytes (if 1 or 2 bytes) with padding */
00112 if (buf_len > 0) {
00113     encode_base64_block(buffer, buf_len, out);
00114 }
00115
00116 /* Optional trailing newline */
00117 fputc('\n', out);
00118
00119 return 0;
00120 }
00121
00122 int hex2b64_buffer(const unsigned char *hex,
00123                     size_t hex_len,
00124                     unsigned char *out,
00125                     size_t out_cap,
00126                     size_t *out_len) {
00127     if (!out || (hex_len > 0 && !hex)) {
00128         return 1;
00129     }
00130
00131     unsigned char buffer[3];
00132     size_t buf_len = 0;
00133     int high_nibble = -1;
00134     size_t produced = 0;
00135
00136     for (size_t i = 0; i < hex_len; ++i) {
00137         unsigned char ch = hex[i];
00138         if (isspace((unsigned char)ch)) {
00139             continue;
00140         }
00141
00142         int v = hex_value(ch);
00143         if (v < 0) {
00144             return 1;
00145         }
00146
00147         if (high_nibble < 0) {
00148             high_nibble = v;
00149         } else {
00150             unsigned char byte = (unsigned char)((high_nibble << 4) | v);
00151             high_nibble = -1;
00152             buffer[buf_len++] = byte;
00153
00154             if (buf_len == 3) {
00155                 if (produced + 4 > out_cap) {
00156                     return 1;
00157                 }
00158                 char encoded[4];
00159                 encode_base64_chars(buffer, buf_len, encoded);
00160                 out[produced++] = (unsigned char)encoded[0];
00161                 out[produced++] = (unsigned char)encoded[1];
00162                 out[produced++] = (unsigned char)encoded[2];
00163                 out[produced++] = (unsigned char)encoded[3];
00164                 buf_len = 0;
00165             }
00166         }
00167     }
00168 }
00169
00170 if (high_nibble >= 0) {
00171     return 1;
00172 }
00173
00174 if (buf_len > 0) {
00175     if (produced + 4 > out_cap) {
00176         return 1;
00177     }
00178     char encoded[4];
00179     encode_base64_chars(buffer, buf_len, encoded);
00180     out[produced++] = (unsigned char)encoded[0];

```

```

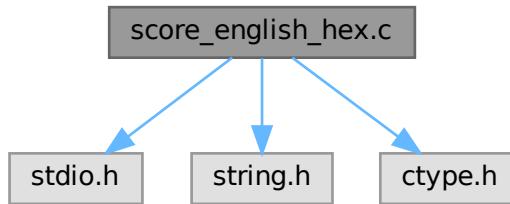
00181     out [produced++] = (unsigned char)encoded[1];
00182     out [produced++] = (unsigned char)encoded[2];
00183     out [produced++] = (unsigned char)encoded[3];
00184 }
00185
00186 if (produced + 1 > out_cap) {
00187     return 1;
00188 }
00189 out [produced++] = '\n';
00190
00191 if (out_len) {
00192     *out_len = produced;
00193 }
00194
00195 return 0;
00196 }
```

3.15 score_english_hex.c File Reference

Implementation of English scoring for hex-encoded strings.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
```

Include dependency graph for score_english_hex.c:



Functions

- static int `hex_value` (char c)
Convert a single hex character to its integer value.
- double `score_english_hex` (const char *hex)
Implementation of `score_english_hex()`.

Variables

- static const double `english_freq` [27]
English letter frequency proportions for a-z plus space.

3.15.1 Detailed Description

Implementation of English scoring for hex-encoded strings.
 Definition in file [score_english_hex.c](#).

3.15.2 Function Documentation

3.15.2.1 hex_value()

```
static int hex_value (
    char c )  [static]
```

Convert a single hex character to its integer value.

Parameters

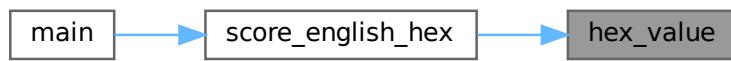
c	Hex digit to convert.
---	-----------------------

Returns

0-15 on success or -1 on invalid characters.

Definition at line 51 of file [score_english_hex.c](#).

Here is the caller graph for this function:



3.15.2.2 score_english_hex()

```
double score_english_hex (
    const char * hex )
```

Implementation of [score_english_hex\(\)](#).

Score a hex string based on English letter frequency heuristics.

Definition at line 59 of file [score_english_hex.c](#).

References [english_freq](#), and [hex_value\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



3.15.3 Variable Documentation

3.15.3.1 english_freq

```
const double english_freq[27] [static]
```

Initial value:

```
= {
    0.0817,
    0.0150,
    0.0278,
    0.0425,
    0.1270,
    0.0223,
    0.0202,
    0.0609,
    0.0697,
    0.0015,
    0.0077,
    0.0403,
    0.0241,
    0.0675,
    0.0751,
    0.0193,
    0.0010,
    0.0599,
    0.0633,
    0.0906,
    0.0276,
    0.0098,
    0.0236,
    0.0015,
    0.0197,
    0.0007,
    0.1300
}
```

English letter frequency proportions for a-z plus space.

Index 0-25 correspond to 'a'-'z' and index 26 represents space.

Definition at line 15 of file [score_english_hex.c](#).

3.16 score_english_hex.c

[Go to the documentation of this file.](#)

```
00001
00006 #include <stdio.h>
00007 #include <string.h>
00008 #include <ctype.h>
00009
00015 static const double english_freq[27] = {
00016     0.0817, // a
00017     0.0150, // b
00018     0.0278, // c
00019     0.0425, // d
00020     0.1270, // e
00021     0.0223, // f
00022     0.0202, // g
00023     0.0609, // h
00024     0.0697, // i
00025     0.0015, // j
00026     0.0077, // k
00027     0.0403, // l
00028     0.0241, // m
00029     0.0675, // n
00030     0.0751, // o
00031     0.0193, // p
00032     0.0010, // q
00033     0.0599, // r
00034     0.0633, // s
00035     0.0906, // t
00036     0.0276, // u
00037     0.0098, // v
00038     0.0236, // w
00039     0.0015, // x
00040     0.0197, // y
00041     0.0007, // z
00042     0.1300 // space (very common in normal English text)
00043 };
00044
00051 static int hex_value(char c) {
00052     if (c >= '0' && c <= '9') return c - '0';
00053     if (c >= 'a' && c <= 'f') return 10 + (c - 'a');
00054     if (c >= 'A' && c <= 'F') return 10 + (c - 'A');
```

```

00055     return -1;
00056 }
00057
00059 double score_english_hex(const char *hex) {
00060     size_t hex_len = strlen(hex);
00061     if (hex_len == 0 || (hex_len & 1)) {
00062         // empty or odd-length hex is invalid
00063         return -1e9;
00064     }
00065
00066     int counts[27] = {0}; // letter+space counts
00067     size_t total_letters = 0;
00068     size_t total_bytes = 0;
00069     double penalty = 0.0;
00070
00071     // Decode hex on the fly; no need to allocate a separate buffer.
00072     for (size_t i = 0; i < hex_len; i += 2) {
00073         int hi = hex_value(hex[i]);
00074         int lo = hex_value(hex[i + 1]);
00075         if (hi < 0 || lo < 0) {
00076             // invalid hex character
00077             return -1e9;
00078         }
00079
00080         unsigned char c = (unsigned char)((hi << 4) | lo);
00081         total_bytes++;
00082
00083         if (isalpha(c)) {
00084             c = (unsigned char)tolower(c);
00085             counts[c - 'a']++;
00086             total_letters++;
00087         } else if (c == ' ') {
00088             counts[26]++;
00089             total_letters++;
00090         } else if (c == '\n' || c == '\r' || c == '\t' ||
00091             c == ',' || c == '.' || c == '\"' || c == "'") {
00092             // Neutral punctuation/whitespace: allowed but not counted as letters.
00093         } else if (c < 32 || c > 126) {
00094             // Non-printable or non-ASCII characters: penalize heavily but keep scoring.
00095             penalty += 50.0;
00096             continue;
00097         } else {
00098             // Other printable symbols like ! ? ; : etc. are allowed but not counted as letters.
00099         }
00100     }
00101
00102     if (total_bytes == 0) {
00103         return -1e9;
00104     }
00105     if (total_letters == 0) {
00106         return -1000.0 - penalty;
00107     }
00108
00109     // Compute a chi-squared style statistic over letters+space.
00110     // Lower chi2 means closer to English; we will invert it into a score.
00111     double chi2 = 0.0;
00112     for (int i = 0; i < 27; i++) {
00113         double expected = english_freq[i] * (double)total_letters;
00114         double observed = (double)counts[i];
00115         double diff = observed - expected;
00116         // Add a tiny constant to avoid division by zero.
00117         chi2 += (diff * diff) / (expected + 1e-9);
00118     }
00119
00120     // Convert chi-squared to a score.
00121     // Smaller chi2 -> higher score. Add bonus for high proportion of letters/spaces.
00122     double letter_ratio = (double)total_letters / (double)total_bytes;
00123     double score = -chi2 + letter_ratio * 50.0 - penalty;
00124
00125     return score;
00126 }

```

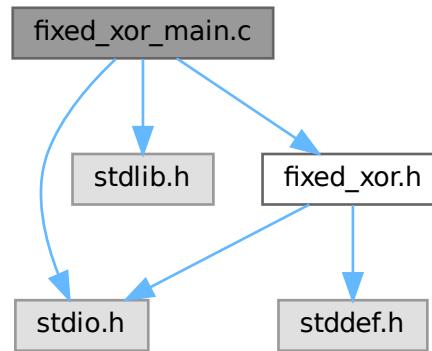
3.17 README.md File Reference

3.18 fixed_xor_main.c File Reference

Command-line wrapper for XORing two equal-length buffers.

```
#include <stdio.h>
#include <stdlib.h>
#include "fixed_xor.h"
```

Include dependency graph for fixed_xor_main.c:



Functions

- int `main` (void)
Entry point that streams stdin through `fixed_xor_stream()`.

3.18.1 Detailed Description

Command-line wrapper for XORing two equal-length buffers.
 Definition in file [fixed_xor_main.c](#).

3.18.2 Function Documentation

3.18.2.1 main()

```
int main (
    void )
```

Entry point that streams stdin through [fixed_xor_stream\(\)](#).

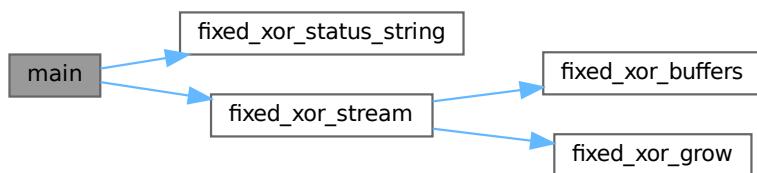
Returns

`EXIT_SUCCESS` on success, `EXIT_FAILURE` otherwise.

Definition at line 16 of file [fixed_xor_main.c](#).

References `FIXED_XOR_OK`, [fixed_xor_status_string\(\)](#), and [fixed_xor_stream\(\)](#).

Here is the call graph for this function:



3.19 fixed_xor_main.c

[Go to the documentation of this file.](#)

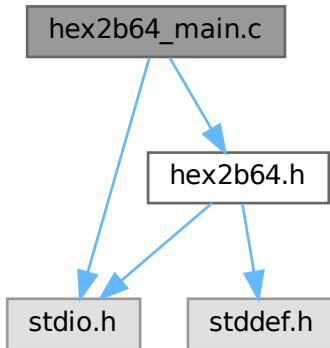
```
00001
00006 #include <stdio.h>
00007 #include <stdlib.h>
00008
00009 #include "fixed_xor.h"
00010
00016 int main(void) {
00017     fixed_xor_status status = fixed_xor_stream(stdin, stdout);
00018     if (status != FIXED_XOR_OK) {
00019         fprintf(stderr, "fixed_xor: %s\n", fixed_xor_status_string(status));
00020         return EXIT_FAILURE;
00021     }
00022
00023     return EXIT_SUCCESS;
00024 }
```

3.20 hex2b64_main.c File Reference

Command-line tool that converts hex input to Base64.

```
#include <stdio.h>
#include "hex2b64.h"
```

Include dependency graph for hex2b64_main.c:



Functions

- int `main` (void)
Entry point that wires stdin/stdout through `hex2b64_stream()`.

3.20.1 Detailed Description

Command-line tool that converts hex input to Base64.

Definition in file [hex2b64_main.c](#).

3.20.2 Function Documentation

3.20.2.1 main()

```
int main (
    void )
```

Entry point that wires stdin/stdout through `hex2b64_stream()`.

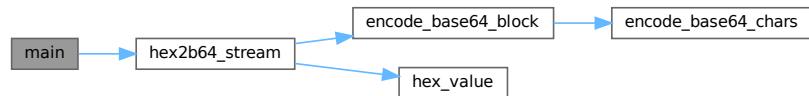
Returns

0 on success, non-zero if conversion fails.

Definition at line 14 of file [hex2b64_main.c](#).

References [hex2b64_stream\(\)](#).

Here is the call graph for this function:



3.21 hex2b64_main.c

[Go to the documentation of this file.](#)

```

00001
00006 #include <stdio.h>
00007 #include "hex2b64.h"
00008
00014 int main(void) {
00015     return hex2b64_stream(stdin, stdout);
00016 }
  
```

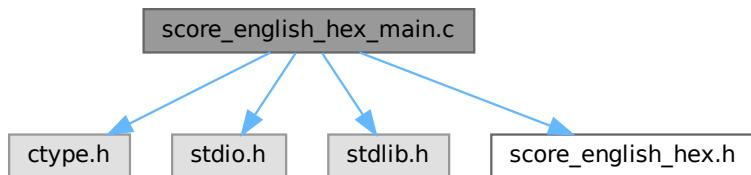
3.22 score_english_hex_main.c File Reference

CLI utility that scores hex data for English-likeness.

```

#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include "score_english_hex.h"
  
```

Include dependency graph for `score_english_hex_main.c`:



Functions

- static double `score_to_percentage` (double raw)
Map a raw English score to a 0-100 percentage.
- int `main` (void)
Read hex from stdin, score it, and print a percentage.

3.22.1 Detailed Description

CLI utility that scores hex data for English-likeness.

Definition in file [score_english_hex_main.c](#).

3.22.2 Function Documentation

3.22.2.1 main()

```
int main (
    void )
```

Read hex from stdin, score it, and print a percentage.

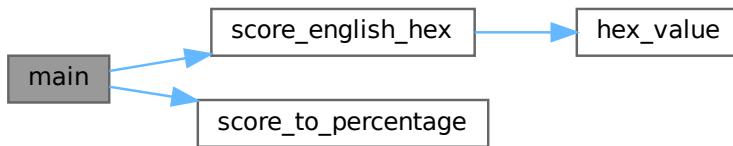
Returns

EXIT_SUCCESS on valid input, EXIT_FAILURE otherwise.

Definition at line 38 of file [score_english_hex_main.c](#).

References [score_english_hex\(\)](#), and [score_to_percentage\(\)](#).

Here is the call graph for this function:



3.22.2.2 score_to_percentage()

```
static double score_to_percentage (
    double raw ) [static]
```

Map a raw English score to a 0-100 percentage.

Parameters

<i>raw</i>	Score returned by score_english_hex() .
------------	---

Returns

Normalized percentage.

Definition at line 18 of file [score_english_hex_main.c](#).

Here is the caller graph for this function:



3.23 score_english_hex_main.c

[Go to the documentation of this file.](#)

```
00001
00006 #include <ctype.h>
00007 #include <stdio.h>
00008 #include <stdlib.h>
00009
00010 #include "score_english_hex.h"
00011
00018 static double score_to_percentage(double raw) {
00019     if (raw < -1e8) {
00020         return 0.0;
00021     }
00022
00023     double normalized = raw + 50.0;
00024     if (normalized < 0.0) {
00025         normalized = 0.0;
00026     }
00027     if (normalized > 100.0) {
00028         normalized = 100.0;
00029     }
00030     return normalized;
00031 }
00032
00038 int main(void) {
00039     size_t capacity = 1024;
00040     size_t len = 0;
00041     char *hex = malloc(capacity);
00042     if (!hex) {
00043         fprintf(stderr, "score_english_hex: failed to allocate input buffer\n");
00044         return EXIT_FAILURE;
00045     }
00046
00047     int ch;
00048     while ((ch = fgetc(stdin)) != EOF) {
00049         if (isspace((unsigned char)ch)) {
00050             continue;
00051         }
00052
00053         if (len + 1 >= capacity) {
00054             size_t new_capacity = capacity * 2;
00055             char *tmp = realloc(hex, new_capacity);
00056             if (!tmp) {
00057                 free(hex);
00058                 fprintf(stderr, "score_english_hex: failed to grow input buffer\n");
00059                 return EXIT_FAILURE;
00060             }
00061             hex = tmp;
00062             capacity = new_capacity;
00063         }
00064
00065         hex[len++] = (char)ch;
00066     }
00067     hex[len] = '\0';
00068
00069     if (len == 0) {
00070         free(hex);
00071         fprintf(stderr, "score_english_hex: no hex data provided\n");
00072         return EXIT_FAILURE;
00073     }
00074
00075     double raw_score = score_english_hex(hex);
00076     free(hex);
00077
00078     if (raw_score < -1e8) {
00079         fprintf(stderr, "score_english_hex: invalid hex input\n");
00080         return EXIT_FAILURE;
00081     }
00082
00083     double percentage = score_to_percentage(raw_score);
00084     printf("%.2f\n", percentage);
00085
00086     return EXIT_SUCCESS;
00087 }
```


Index

b64_table
 hex2b64.c, 27

Cryptopals in C Code, 1

encode_base64_block
 hex2b64.c, 23

encode_base64_chars
 hex2b64.c, 24

english_freq
 score_english_hex.c, 31

fixed_xor.c, 18, 21
 fixed_xor_buffers, 19
 FIXED_XOR_CHUNK, 19
 fixed_xor_grow, 19
 fixed_xor_status_string, 20
 fixed_xor_stream, 20

fixed_xor.h, 9, 13
 fixed_xor_buffers, 11
 FIXED_XOR_ERR_ARGS, 11
 FIXED_XOR_ERR_IO, 11
 FIXED_XOR_ERR_ODD_INPUT, 11
 FIXED_XOR_ERR_OOM, 11
 FIXED_XOR_OK, 11
 fixed_xor_status, 11
 fixed_xor_status_string, 11
 fixed_xor_stream, 12

fixed_xor_buffers
 fixed_xor.c, 19
 fixed_xor.h, 11

FIXED_XOR_CHUNK
 fixed_xor.c, 19

FIXED_XOR_ERR_ARGS
 fixed_xor.h, 11

FIXED_XOR_ERR_IO
 fixed_xor.h, 11

FIXED_XOR_ERR_ODD_INPUT
 fixed_xor.h, 11

FIXED_XOR_ERR_OOM
 fixed_xor.h, 11

fixed_xor_grow
 fixed_xor.c, 19

fixed_xor_main.c, 32, 34
 main, 33

FIXED_XOR_OK
 fixed_xor.h, 11

fixed_xor_status
 fixed_xor.h, 11

fixed_xor_status_string

fixed_xor.c, 20
fixed_xor.h, 11

fixed_xor_stream
 fixed_xor.c, 20
 fixed_xor.h, 12

hex2b64.c, 23, 27
 b64_table, 27
 encode_base64_block, 23
 encode_base64_chars, 24
 hex2b64_buffer, 25
 hex2b64_stream, 25
 hex_value, 26

hex2b64.h, 13, 16
 hex2b64_buffer, 14
 hex2b64_stream, 15

hex2b64_buffer
 hex2b64.c, 25
 hex2b64.h, 14

hex2b64_main.c, 34, 35
 main, 34

hex2b64_stream
 hex2b64.c, 25
 hex2b64.h, 15

hex_digit_value
 set_1_challenge_2.c, 7

hex_to_bytes
 set_1_challenge_2.c, 7

hex_value
 hex2b64.c, 26
 score_english_hex.c, 29

main
 fixed_xor_main.c, 33
 hex2b64_main.c, 34
 score_english_hex_main.c, 36
 set_1_challenge_1.c, 5
 set_1_challenge_2.c, 8

README.md, 32

score_english_hex
 score_english_hex.c, 30
 score_english_hex.h, 17

score_english_hex.c, 29, 31
 english_freq, 31
 hex_value, 29
 score_english_hex, 30

score_english_hex.h, 16, 18
 score_english_hex, 17

score_english_hex_main.c, 35, 36
 main, 36
 score_to_percentage, 36
score_to_percentage
 score_english_hex_main.c, 36
set_1_challenge_1.c, 5, 6
 main, 5
set_1_challenge_2.c, 6, 8
 hex_digit_value, 7
 hex_to_bytes, 7
 main, 8