

The DES library.

Please note that this library was originally written to operate with eBones, a version of Kerberos that had had encryption removed when it left the USA and then put back in. As such there are some routines that I will advise not using but they are still in the library for historical reasons. For all calls that have an 'input' and 'output' variables, they can be the same.

This library requires the inclusion of 'des.h'.

All of the encryption functions take what is called a `des_key_schedule` as an argument. A `des_key_schedule` is an expanded form of the des key. A `des_key` is 8 bytes of odd parity, the type used to hold the key is a `des_cblock`. A `des_cblock` is an array of 8 bytes, often in this library description I will refer to input bytes when the function specifies `des_cblock`'s as input or output, this just means that the variable should be a multiple of 8 bytes.

The define `DES_ENCRYPT` is passed to specify encryption, `DES_DECRYPT` to specify decryption. The functions and global variable are as follows:

```
int des_check_key;
```

DES keys are supposed to be odd parity. If this variable is set to a non-zero value, `des_set_key()` will check that the key has odd parity and is not one of the known weak DES keys. By default this variable is turned off;

```
void des_set_odd_parity(
des_cblock *key );
```

This function takes a DES key (8 bytes) and sets the parity to odd.

```
int des_is_weak_key(
des_cblock *key );
```

This function returns a non-zero value if the DES key passed is a weak, DES key. If it is a weak key, don't use it, try a different one. If you are using 'random' keys, the chances of hitting a weak key are $1/2^{52}$ so it is probably not worth checking for them.

```
int des_set_key(
des_cblock *key,
des_key_schedule schedule);
```

`Des_set_key` converts an 8 byte DES key into a `des_key_schedule`. A `des_key_schedule` is an expanded form of the key which is used to perform actual encryption. It can be regenerated from the DES key so it only needs to be kept when encryption or decryption is about to occur. Don't save or pass around `des_key_schedule`'s since they are CPU architecture dependent, DES keys are not. If `des_check_key` is non zero, zero is returned if the key has the wrong parity or the key is a weak key, else 1 is returned.

```
int des_key_sched(
des_cblock *key,
des_key_schedule schedule);
```

An alternative name for `des_set_key()`.

```
int des_rw_mode;          /* defaults to DES_PCBC_MODE */
```

This flag holds either `DES_CBC_MODE` or `DES_PCBC_MODE` (default).

This specifies the function to use in the enc_read() and enc_write() functions.

```
void des_encrypt(
unsigned long *data,
des_key_schedule ks,
int enc);
```

This is the DES encryption function that gets called by just about every other DES routine in the library. You should not use this function except to implement 'modes' of DES. I say this because the functions that call this routine do the conversion from 'char *' to long, and this needs to be done to make sure 'non-aligned' memory access do not occur. The characters are loaded 'little endian', have a look at my source code for more details on how I use this function.

Data is a pointer to 2 unsigned long's and ks is the des_key_schedule to use. enc, is non zero specifies encryption, zero if decryption.

```
void des_encrypt2(
unsigned long *data,
des_key_schedule ks,
int enc);
```

This functions is the same as des_encrypt() except that the DES initial permutation (IP) and final permutation (FP) have been left out. As for des_encrypt(), you should not use this function.

It is used by the routines in my library that implement triple DES. IP() des_encrypt2() des_encrypt2() des_encrypt2() FP() is the same as des_encrypt() des_encrypt() des_encrypt() except faster :-).

```
void des_ecb_encrypt(
des_cblock *input,
des_cblock *output,
des_key_schedule ks,
int enc);
```

This is the basic Electronic Code Book form of DES, the most basic form. Input is encrypted into output using the key represented by ks. If enc is non zero (DES_ENCRYPT), encryption occurs, otherwise decryption occurs. Input is 8 bytes long and output is 8 bytes. (the des_cblock structure is 8 chars).

```
void des_ecb3_encrypt(
des_cblock *input,
des_cblock *output,
des_key_schedule ks1,
des_key_schedule ks2,
des_key_schedule ks3,
int enc);
```

This is the 3 key EDE mode of ECB DES. What this means is that the 8 bytes of input is encrypted with ks1, decrypted with ks2 and then encrypted again with ks3, before being put into output; $C = E(ks3, D(ks2, E(ks1, M)))$. There is a macro, des_ecb2_encrypt() that only takes 2 des_key_schedules that implements, $C = E(ks1, D(ks2, E(ks1, M)))$ in that the final encrypt is done with ks1.

```
void des_cbc_encrypt(
des_cblock *input,
des_cblock *output,
```

```
long length,  
des_key_schedule ks,  
des_cblock *ivec,  
int enc);
```

This routine implements DES in Cipher Block Chaining mode. Input, which should be a multiple of 8 bytes is encrypted (or decrypted) to output which will also be a multiple of 8 bytes. The number of bytes is in length (and from what I've said above, should be a multiple of 8). If length is not a multiple of 8, I'm not being held responsible :-). ivec is the initialisation vector. This function does not modify this variable. To correctly implement cbc mode, you need to do one of 2 things; copy the last 8 bytes of cipher text for use as the next ivec in your application, or use `des_ncbc_encrypt()`. Only this routine has this problem with updating the ivec, all other routines that are implementing cbc mode update ivec.

```
void des_ncbc_encrypt(  
des_cblock *input,  
des_cblock *output,  
long length,  
des_key_schedule sk,  
des_cblock *ivec,  
int enc);
```

For historical reasons, `des_cbc_encrypt()` did not update the ivec with the value requires so that subsequent calls to `des_cbc_encrypt()` would 'chain'. This was needed so that the same 'length' values would not need to be used when decrypting. `des_ncbc_encrypt()` does the right thing. It is the same as `des_cbc_encrypt` accept that ivec is updates with the correct value to pass in subsequent calls to `des_ncbc_encrypt()`. I advise using `des_ncbc_encrypt()` instead of `des_cbc_encrypt()`;

```
void des_xcbc_encrypt(  
des_cblock *input,  
des_cblock *output,  
long length,  
des_key_schedule sk,  
des_cblock *ivec,  
des_cblock *inw,  
des_cblock *outw,  
int enc);
```

This is RSA's DESX mode of DES. It uses inw and outw to 'whiten' the encryption. inw and outw are secret (unlike the iv) and are as such, part of the key. So the key is sort of 24 bytes. This is much better than cbc des.

```
void des_3cbc_encrypt(  
des_cblock *input,  
des_cblock *output,  
long length,  
des_key_schedule sk1,  
des_key_schedule sk2,  
des_cblock *ivec1,  
des_cblock *ivec2,  
int enc);
```

This function is flawed, do not use it. I have left it in the library because it is used in my `des(1)` program and will function

correctly when used by des(1). If I removed the function, people could end up unable to decrypt files.
This routine implements outer triple cbc encryption using 2 ks and 2 ivec's. Use des_edc2_cbc_encrypt() instead.

```
void des_edc3_cbc_encrypt(
des_cblock *input,
des_cblock *output,
long length,
des_key_schedule ks1,
des_key_schedule ks2,
des_key_schedule ks3,
des_cblock *ivec,
int enc);
```

This function implements inner triple CBC DES encryption with 3 keys. What this means is that each 'DES' operation inside the cbc mode is really an $C=E(ks3,D(ks2,E(ks1,M)))$. Again, this is cbc mode so an ivec is required. This mode is used by SSL.
There is also a des_edc2_cbc_encrypt() that only uses 2 des_key_schedule's, the first being reused for the final encryption. $C=E(ks1,D(ks2,E(ks1,M)))$. This form of triple DES is used by the RSAREF library.

```
void des_pcbc_encrypt(
des_cblock *input,
des_cblock *output,
long length,
des_key_schedule ks,
des_cblock *ivec,
int enc);
```

This is Propagating Cipher Block Chaining mode of DES. It is used by Kerberos v4. Its parameters are the same as des_ncbc_encrypt().

```
void des_cfb_encrypt(
unsigned char *in,
unsigned char *out,
int numbits,
long length,
des_key_schedule ks,
des_cblock *ivec,
int enc);
```

Cipher Feedback mode of DES. This implementation 'feeds back' in numbit blocks. The input (and output) is in multiples of numbits bits. numbits should be a multiple of 8 bits. Length is the number of bytes input. If numbits is not a multiple of 8 bits, the extra bits in the bytes will be considered padding. So if numbits is 12, for each 2 input bytes, the 4 high bits of the second byte will be ignored. So to encode 72 bits when using a numbits of 12 take 12 bytes. To encode 72 bits when using numbits of 9 will take 16 bytes. To encode 80 bits when using numbits of 16 will take 10 bytes. etc, etc. This padding will apply to both input and output.

```
void des_cfb64_encrypt(
unsigned char *in,
unsigned char *out,
```

```

long length,
des_key_schedule ks,
des_cblock *ivec,
int *num,
int enc);

```

This is one of the more useful functions in this DES library, it implements CFB mode of DES with 64bit feedback. Why is this useful you ask? Because this routine will allow you to encrypt an arbitrary number of bytes, no 8 byte padding. Each call to this routine will encrypt the input bytes to output and then update ivec and num. num contains 'how far' we are though ivec. If this does not make much sense, read more about cfb mode of DES :-).

```

void des_edes3_cfb64_encrypt(
unsigned char *in,
unsigned char *out,
long length,
des_key_schedule ks1,
des_key_schedule ks2,
des_key_schedule ks3,
des_cblock *ivec,
int *num,
int enc);

```

Same as des_cfb64_encrypt() accept that the DES operation is triple DES. As usual, there is a macro for des_edes2_cfb64_encrypt() which reuses ks1.

```

void des_ofb_encrypt(
unsigned char *in,
unsigned char *out,
int numbits,
long length,
des_key_schedule ks,
des_cblock *ivec);

```

This is a implementation of Output Feed Back mode of DES. It is the same as des_cfb_encrypt() in that numbits is the size of the units dealt with during input and output (in bits).

```

void des_ofb64_encrypt(
unsigned char *in,
unsigned char *out,
long length,
des_key_schedule ks,
des_cblock *ivec,
int *num);

```

The same as des_cfb64_encrypt() except that it is Output Feed Back mode.

```

void des_edes3_ofb64_encrypt(
unsigned char *in,
unsigned char *out,
long length,
des_key_schedule ks1,
des_key_schedule ks2,
des_key_schedule ks3,
des_cblock *ivec,
int *num);

```

Same as des_ofb64_encrypt() accept that the DES operation is

triple DES. As usual, there is a macro for `des_ede2_ofb64_encrypt()` which reuses `ks1`.

```
int des_read_pw_string(
char *buf,
int length,
char *prompt,
int verify);
```

This routine is used to get a password from the terminal with echo turned off. `Buf` is where the string will end up and `length` is the size of `buf`. `Prompt` is a string presented to the 'user' and if `verify` is set, the key is asked for twice and unless the 2 copies match, an error is returned. A return code of -1 indicates a system error, 1 failure due to user interaction, and 0 is success.

```
unsigned long des_cbc_cksum(
des_cblock *input,
des_cblock *output,
long length,
des_key_schedule ks,
des_cblock *ivec);
```

This function produces an 8 byte checksum from input that it puts in output and returns the last 4 bytes as a long. The checksum is generated via cbc mode of DES in which only the last 8 bytes are kept. I would recommend not using this function but instead using the `EVP_Digest` routines, or at least using MD5 or SHA. This function is used by Kerberos v4 so that is why it stays in the library.

```
char *des_fcrypt(
const char *buf,
const char *salt
char *ret);
```

This is my fast version of the `unix crypt(3)` function. This version takes only a small amount of space relative to other fast `crypt()` implementations. This is different to the normal `crypt` in that the third parameter is the buffer that the return value is written into. It needs to be at least 14 bytes long. This function is thread safe, unlike the normal `crypt`.

```
char *crypt(
const char *buf,
const char *salt);
```

This function calls `des_fcrypt()` with a static array passed as the third parameter. This emulates the normal non-thread safe semantics of `crypt(3)`.

```
void des_string_to_key(
char *str,
des_cblock *key);
```

This function takes `str` and converts it into a DES key. I would recommend using MD5 instead and use the first 8 bytes of output. When I wrote the first version of these routines back in 1990, MD5 did not exist but I feel these routines are still sound. This routines is compatible with the one in MIT's `libdes`.

```
void des_string_to_2keys(
char *str,
```

```
des_cblock *key1,  
des_cblock *key2);
```

This function takes str and converts it into 2 DES keys.
I would recommend using MD5 and using the 16 bytes as the 2 keys.
I have nothing against these 2 'string_to_key' routines, it's just that if you say that your encryption key is generated by using the 16 bytes of an MD5 hash, every-one knows how you generated your keys.

```
int des_read_password(  
des_cblock *key,  
char *prompt,  
int verify);
```

This routine combines des_read_pw_string() with des_string_to_key().

```
int des_read_2passwords(  
des_cblock *key1,  
des_cblock *key2,  
char *prompt,  
int verify);
```

This routine combines des_read_pw_string() with des_string_to_2key().

```
void des_random_seed(  
des_cblock key);
```

This routine sets a starting point for des_random_key().

```
void des_random_key(  
des_cblock ret);
```

This function return a random key. Make sure to 'seed' the random number generator (with des_random_seed()) before using this function. I personally now use a MD5 based random number system.

```
int des_enc_read(  
int fd,  
char *buf,  
int len,  
des_key_schedule ks,  
des_cblock *iv);
```

This function will write to a file descriptor the encrypted data from buf. This data will be preceded by a 4 byte 'byte count' and will be padded out to 8 bytes. The encryption is either CBC or PCBC depending on the value of des_rw_mode. If it is DES_PCBC_MODE, pcbc is used, if DES_CBC_MODE, cbc is used. The default is to use DES_PCBC_MODE.

```
int des_enc_write(  
int fd,  
char *buf,  
int len,  
des_key_schedule ks,  
des_cblock *iv);
```

This routines read stuff written by des_enc_read() and decrypts it. I have used these routines quite a lot but I don't believe they are suitable for non-blocking io. If you are after a full authentication/encryption over networks, have a look at SSL instead.

```
unsigned long des_quad_cksum(  
des_cblock *input,
```

```
des_cblock *output,  
long length,  
int out_count,  
des_cblock *seed);  
    This is a function from Kerberos v4 that is not anything to do with  
    DES but was needed. It is a cksum that is quicker to generate than  
    des_cbc_cksum(); I personally would use MD5 routines now.
```

=====

Modes of DES

Quite a bit of the following information has been taken from

AS 2805.5.2

Australian Standard

Electronic funds transfer - Requirements for interfaces,

Part 5.2: Modes of operation for an n-bit block cipher algorithm

Appendix A

There are several different modes in which DES can be used, they are as follows.

Electronic Codebook Mode (ECB) (des_ecb_encrypt())

- 64 bits are enciphered at a time.
- The order of the blocks can be rearranged without detection.
- The same plaintext block always produces the same ciphertext block (for the same key) making it vulnerable to a 'dictionary attack'.
- An error will only affect one ciphertext block.

Cipher Block Chaining Mode (CBC) (des_cbc_encrypt())

- a multiple of 64 bits are enciphered at a time.
- The CBC mode produces the same ciphertext whenever the same plaintext is encrypted using the same key and starting variable.
- The chaining operation makes the ciphertext blocks dependent on the current and all preceding plaintext blocks and therefore blocks can not be rearranged.
- The use of different starting variables prevents the same plaintext enciphering to the same ciphertext.
- An error will affect the current and the following ciphertext blocks.

Cipher Feedback Mode (CFB) (des_cfb_encrypt())

- a number of bits (j) \leq 64 are enciphered at a time.
- The CFB mode produces the same ciphertext whenever the same plaintext is encrypted using the same key and starting variable.
- The chaining operation makes the ciphertext variables dependent on the current and all preceding variables and therefore j-bit variables are chained together and can not be rearranged.
- The use of different starting variables prevents the same plaintext enciphering to the same ciphertext.
- The strength of the CFB mode depends on the size of k (maximal if $j = k$). In my implementation this is always the case.
- Selection of a small value for j will require more cycles through the encipherment algorithm per unit of plaintext and thus cause greater processing overheads.
- Only multiples of j bits can be enciphered.
- An error will affect the current and the following ciphertext variables.

Output Feedback Mode (OFB) (des_ofb_encrypt())

- a number of bits (j) \leq 64 are enciphered at a time.
- The OFB mode produces the same ciphertext whenever the same plaintext enciphered using the same key and starting variable. More

over, in the OFB mode the same key stream is produced when the same key and start variable are used. Consequently, for security reasons a specific start variable should be used only once for a given key.

- The absence of chaining makes the OFB more vulnerable to specific attacks.
- The use of different start variables values prevents the same plaintext enciphering to the same ciphertext, by producing different key streams.
- Selection of a small value for j will require more cycles through the encipherment algorithm per unit of plaintext and thus cause greater processing overheads.
- Only multiples of j bits can be enciphered.
- OFB mode of operation does not extend ciphertext errors in the resultant plaintext output. Every bit error in the ciphertext causes only one bit to be in error in the deciphered plaintext.
- OFB mode is not self-synchronising. If the two operation of encipherment and decipherment get out of synchronism, the system needs to be re-initialised.
- Each re-initialisation should use a value of the start variable different from the start variable values used before with the same key. The reason for this is that an identical bit stream would be produced each time from the same parameters. This would be susceptible to a 'known plaintext' attack.

Triple ECB Mode (`des_ecb3_encrypt()`)

- Encrypt with key1, decrypt with key2 and encrypt with key3 again.
- As for ECB encryption but increases the key length to 168 bits. There are theoretic attacks that can be used that make the effective key length 112 bits, but this attack also requires 2^{56} blocks of memory, not very likely, even for the NSA.
- If both keys are the same it is equivalent to encrypting once with just one key.
- If the first and last key are the same, the key length is 112 bits. There are attacks that could reduce the key space to 55 bit's but it requires 2^{56} blocks of memory.
- If all 3 keys are the same, this is effectively the same as normal ecb mode.

Triple CBC Mode (`des_ede3_cbc_encrypt()`)

- Encrypt with key1, decrypt with key2 and then encrypt with key3.
- As for CBC encryption but increases the key length to 168 bits with the same restrictions as for triple ecb mode.