NAME

EVP_CIPHER_CTX_init, EVP_EncryptInit_ex, EVP_EncryptUpdate, EVP_EncryptFinal_ex, EVP_DecryptInit_ex, EVP_DecryptUpdate, EVP_DecryptFinal_ex, EVP_CipherInit_ex, EVP_CipherUpdate, EVP_CipherFinal_ex, EVP_CIPHER_CTX_set_key_length, EVP_CIPHER_CTX_ctrl, EVP_CIPHER_CTX_cleanup, EVP_EncryptInit, EVP_EncryptFinal, EVP_DecryptInit, EVP_DecryptFinal, EVP_CipherInit, EVP_CipherFinal, EVP_get_cipherbyname, EVP_get_cipherbynid, EVP_get_cipherbyobj, EVP_CIPHER_nid, EVP_CIPHER_block_size, EVP_CIPHER_key_length, EVP_CIPHER_iv_length, EVP_CIPHER_flags, EVP_CIPHER_mode, EVP_CIPHER_type, EVP_CIPHER_CTX_cipher, EVP_CIPHER_CTX_nid, EVP_CIPHER_CTX_block_size, EVP_CIPHER_CTX_key_length, EVP_CIPHER_CTX_iv_length, EVP_CIPHER_CTX_get_app_data, EVP_CIPHER_CTX_set_app_data, EVP_CIPHER_CTX_type, EVP_CIPHER_CTX_flags, EVP_CIPHER_CTX_mode, EVP_CIPHER_param_to_asn1, EVP_CIPHER_asn1_to_param, EVP_CIPHER_CTX_set_padding, EVP_enc_null, EVP_des_cbc, EVP_des_ecb, EVP_des_cfb, EVP_des_ofb, EVP_des_ede_cbc, EVP_des_ede, EVP_des_ede_ofb, EVP_des_ede_cfb, EVP_des_ede3_cbc, EVP_des_ede3, EVP_des_ede3_ofb, EVP_des_ede3_cfb, EVP_desx_cbc, EVP_rc4, EVP_rc4_40, EVP_idea_cbc, EVP_idea_ecb, EVP_idea_cfb, EVP_idea_ofb, EVP_idea_cbc, EVP_rc2_cbc, EVP_rc2_ecb, EVP_rc2_cfb, EVP_rc2_ofb, EVP_rc2_40_cbc, EVP_rc2_64_cbc, EVP_bf_cbc, EVP_bf_ecb, EVP_bf_cfb, EVP_bf_ofb, EVP_cast5_cbc, EVP_cast5_ecb, EVP_cast5_cfb, EVP_cast5_ofb, EVP_rc5_32_12_16_cbc, EVP_rc5_32_12_16_ecb, EVP_rc5_32_12_16_cfb, EVP_rc5_32_12_16_ofb, EVP_aes_128_gcm, EVP_aes_192_gcm, EVP_aes_256_gcm, EVP_aes_128_ccm, EVP_aes_192_ccm, EVP_aes_256_ccm - EVP cipher routines

SYNOPSIS

```
#include <openssl/evp.h>

void EVP_CIPHER_CTX_init(EVP_CIPHER_CTX *a);

int EVP_EncryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
ENGINE *impl, unsigned char *key, unsigned char *iv);
int EVP_EncryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
int *outl, unsigned char *in, int inl);
int EVP_EncryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *out,
int *outl);

int EVP_DecryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
ENGINE *impl, unsigned char *key, unsigned char *iv);
int EVP_DecryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
int *outl, unsigned char *in, int inl);
int EVP_DecryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *outm,
int *outl);

int EVP_CipherInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
ENGINE *impl, unsigned char *key, unsigned char *iv, int enc);
int EVP_CipherUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
int *outl, unsigned char *in, int inl);
int EVP_CipherFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *outm,
int *outl);

int EVP_EncryptInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
unsigned char *key, unsigned char *iv);
int EVP_EncryptFinal(EVP_CIPHER_CTX *ctx, unsigned char *out,
int *outl);
```

```
int EVP_DecryptInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
unsigned char *key, unsigned char *iv);
int EVP_DecryptFinal(EVP_CIPHER_CTX *ctx, unsigned char *outm,
int *outl);

int EVP_CipherInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
unsigned char *key, unsigned char *iv, int enc);
int EVP_CipherFinal(EVP_CIPHER_CTX *ctx, unsigned char *outm,
int *outl);

int EVP_CIPHER_CTX_set_padding(EVP_CIPHER_CTX *x, int padding);
int EVP_CIPHER_CTX_set_key_length(EVP_CIPHER_CTX *x, int keylen);
int EVP_CIPHER_CTX_ctrl(EVP_CIPHER_CTX *ctx, int type, int arg, void *ptr);
int EVP_CIPHER_CTX_cleanup(EVP_CIPHER_CTX *a);

const EVP_CIPHER *EVP_get_cipherbyname(const char *name);
#define EVP_get_cipherbynid(a) EVP_get_cipherbyname(OBJ_nid2sn(a))
#define EVP_get_cipherbyobj(a) EVP_get_cipherbynid(OBJ_obj2nid(a))

#define EVP_CIPHER_nid(e) ((e)->nid)
#define EVP_CIPHER_block_size(e) ((e)->block_size)
#define EVP_CIPHER_key_length(e) ((e)->key_len)
#define EVP_CIPHER_iv_length(e) ((e)->iv_len)
#define EVP_CIPHER_flags(e) ((e)->flags)
#define EVP_CIPHER_mode(e) ((e)->flags) & EVP_CIPH_MODE)
int EVP_CIPHER_type(const EVP_CIPHER *ctx);

#define EVP_CIPHER_CTX_cipher(e) ((e)->cipher)
#define EVP_CIPHER_CTX_nid(e) ((e)->cipher->nid)
#define EVP_CIPHER_CTX_block_size(e) ((e)->cipher->block_size)
#define EVP_CIPHER_CTX_key_length(e) ((e)->key_len)
#define EVP_CIPHER_CTX_iv_length(e) ((e)->cipher->iv_len)
#define EVP_CIPHER_CTX_get_app_data(e) ((e)->app_data)
#define EVP_CIPHER_CTX_set_app_data(e,d) ((e)->app_data=(char *)(d))
#define EVP_CIPHER_CTX_type(c) EVP_CIPHER_type(EVP_CIPHER_CTX_cipher(c))
#define EVP_CIPHER_CTX_flags(e) ((e)->cipher->flags)
#define EVP_CIPHER_CTX_mode(e) ((e)->cipher->flags & EVP_CIPH_MODE)

int EVP_CIPHER_param_to_asn1(EVP_CIPHER_CTX *c, ASN1_TYPE *type);
int EVP_CIPHER_asn1_to_param(EVP_CIPHER_CTX *c, ASN1_TYPE *type);
```

DESCRIPTION

The EVP cipher routines are a high level interface to certain symmetric ciphers.

EVP_CIPHER_CTX_init() initializes cipher contex **ctx**.

EVP_EncryptInit_ex() sets up cipher context **ctx** for encryption with cipher **type** from ENGINE **impl**. **ctx** must be initialized before calling this function. **type** is normally supplied by a function such as *EVP_des_cbc()*. If **impl** is NULL then the default implementation is used. **key** is the symmetric key to use and **iv** is the IV to use (if necessary), the actual number of bytes used for the key and IV depends on the cipher. It is possible to set all parameters to NULL except **type** in an initial call and supply the remaining parameters in subsequent calls, all of which have **type** set to NULL. This is done when the default cipher parameters are not appropriate.

EVP_EncryptUpdate() encrypts **inl** bytes from the buffer **in** and writes the encrypted version to **out**. This function can be called multiple times to encrypt successive blocks of data. The amount of data written depends on the block alignment of the encrypted data: as a result the amount of

data written may be anything from zero bytes to (inl + cipher_block_size - 1) so **out** should contain sufficient room. The actual number of bytes written is placed in **outl**.

If padding is enabled (the default) then *EVP_EncryptFinal_ex()* encrypts the "final" data, that is any data that remains in a partial block. It uses standard block padding (aka PKCS padding). The encrypted final data is written to **out** which should have sufficient space for one cipher block. The number of bytes written is placed in **outl**. After this function is called the encryption operation is finished and no further calls to *EVP_EncryptUpdate()* should be made.

If padding is disabled then *EVP_EncryptFinal_ex()* will not encrypt any more data and it will return an error if any data remains in a partial block: that is if the total data length is not a multiple of the block size.

EVP_DecryptInit_ex(), *EVP_DecryptUpdate()* and *EVP_DecryptFinal_ex()* are the corresponding decryption operations. *EVP_DecryptFinal()* will return an error code if padding is enabled and the final block is not correctly formatted. The parameters and restrictions are identical to the encryption operations except that if padding is enabled the decrypted data buffer **out** passed to *EVP_DecryptUpdate()* should have sufficient room for (**inl** + cipher_block_size) bytes unless the cipher block size is 1 in which case **inl** bytes is sufficient.

EVP_CipherInit_ex(), *EVP_CipherUpdate()* and *EVP_CipherFinal_ex()* are functions that can be used for decryption or encryption. The operation performed depends on the value of the **enc** parameter. It should be set to 1 for encryption, 0 for decryption and -1 to leave the value unchanged (the actual value of 'enc' being supplied in a previous call).

EVP_CIPHER_CTX_cleanup() clears all information from a cipher context and free up any allocated memory associate with it. It should be called after all operations using a cipher are complete so sensitive information does not remain in memory.

EVP_EncryptInit(), *EVP_DecryptInit()* and *EVP_CipherInit()* behave in a similar way to *EVP_EncryptInit_ex()*, EVP_DecryptInit_ex and *EVP_CipherInit_ex()* except the **ctx** parameter does not need to be initialized and they always use the default cipher implementation.

EVP_EncryptFinal(), *EVP_DecryptFinal()* and *EVP_CipherFinal()* behave in a similar way to *EVP_EncryptFinal_ex()*, *EVP_DecryptFinal_ex()* and *EVP_CipherFinal_ex()* except **ctx** is automatically cleaned up after the call.

EVP_get_cipherbyname(), *EVP_get_cipherbynid()* and *EVP_get_cipherbyobj()* return an EVP_CIPHER structure when passed a cipher name, a NID or an ASN1_OBJECT structure.

EVP_CIPHER_nid() and *EVP_CIPHER_CTX_nid()* return the NID of a cipher when passed an **EVP_CIPHER** or **EVP_CIPHER_CTX** structure. The actual NID value is an internal value which may not have a corresponding OBJECT IDENTIFIER.

EVP_CIPHER_CTX_set_padding() enables or disables padding. By default encryption operations are padded using standard block padding and the padding is checked and removed when decrypting. If the **pad** parameter is zero then no padding is performed, the total amount of data encrypted or decrypted must then be a multiple of the block size or an error will occur.

EVP_CIPHER_key_length() and *EVP_CIPHER_CTX_key_length()* return the key length of a cipher when passed an **EVP_CIPHER** or **EVP_CIPHER_CTX** structure. The constant **EVP_MAX_KEY_LENGTH** is the maximum key length for all ciphers. Note: although *EVP_CIPHER_key_length()* is fixed for a given cipher, the value of *EVP_CIPHER_CTX_key_length()* may be different for variable key length ciphers.

EVP_CIPHER_CTX_set_key_length() sets the key length of the cipher ctx. If the cipher is a fixed length cipher then attempting to set the key length to any value other than the fixed value is an error.

EVP_CIPHER_iv_length() and *EVP_CIPHER_CTX_iv_length()* return the IV length of a cipher when passed an **EVP_CIPHER** or **EVP_CIPHER_CTX**. It will return zero if the cipher does not use an IV. The constant **EVP_MAX_IV_LENGTH** is the maximum IV length for all ciphers.

EVP_CIPHER_block_size() and *EVP_CIPHER_CTX_block_size()* return the block size of a cipher when passed an **EVP_CIPHER** or **EVP_CIPHER_CTX** structure. The constant **EVP_MAX_IV_LENGTH** is also the maximum block length for all ciphers.

EVP_CIPHER_type() and *EVP_CIPHER_CTX_type()* return the type of the passed cipher or context. This ''type'' is the actual NID of the cipher OBJECT IDENTIFIER as such it ignores the cipher parameters and 40 bit RC2 and 128 bit RC2 have the same NID. If the cipher does not have an object identifier or does not have ASN1 support this function will return **NID_undef**.

EVP_CIPHER_CTX_cipher() returns the **EVP_CIPHER** structure when passed an **EVP_CIPHER_CTX** structure.

EVP_CIPHER_mode() and *EVP_CIPHER_CTX_mode()* return the block cipher mode: EVP_CIPH_ECB_MODE, EVP_CIPH_CBC_MODE, EVP_CIPH_CFB_MODE or EVP_CIPH_OFB_MODE. If the cipher is a stream cipher then EVP_CIPH_STREAM_CIPHER is returned.

EVP_CIPHER_param_to_asn1() sets the AlgorithmIdentifier ''parameter'' based on the passed cipher. This will typically include any parameters and an IV. The cipher IV (if any) must be set when this call is made. This call should be made before the cipher is actually ''used'' (before any *EVP_EncryptUpdate()*, *EVP_DecryptUpdate()* calls for example). This function may fail if the cipher does not have any ASN1 support.

EVP_CIPHER_asn1_to_param() sets the cipher parameters based on an ASN1 AlgorithmIdentifier ''parameter''. The precise effect depends on the cipher In the case of RC2, for example, it will set the IV and effective key length. This function should be called after the base cipher type is set but before the key is set. For example *EVP_CipherInit()* will be called with the IV and key set to NULL, *EVP_CIPHER_asn1_to_param()* will be called and finally *EVP_CipherInit()* again with all parameters except the key set to NULL. It is possible for this function to fail if the cipher does not have any ASN1 support or the parameters cannot be set (for example the RC2 effective key length is not supported.

EVP_CIPHER_CTX_ctrl() allows various cipher specific parameters to be determined and set.

RETURN VALUES

EVP_EncryptInit_ex(), *EVP_EncryptUpdate()* and *EVP_EncryptFinal_ex()* return 1 for success and 0 for failure.

EVP_DecryptInit_ex() and *EVP_DecryptUpdate()* return 1 for success and 0 for failure. *EVP_DecryptFinal_ex()* returns 0 if the decrypt failed or 1 for success.

EVP_CipherInit_ex() and *EVP_CipherUpdate()* return 1 for success and 0 for failure. *EVP_CipherFinal_ex()* returns 0 for a decryption failure or 1 for success.

EVP_CIPHER_CTX_cleanup() returns 1 for success and 0 for failure.

EVP_get_cipherbyname(), *EVP_get_cipherbynid()* and *EVP_get_cipherbyobj()* return an **EVP_CIPHER** structure or NULL on error.

EVP_CIPHER_nid() and *EVP_CIPHER_CTX_nid()* return a NID.

EVP_CIPHER_block_size() and *EVP_CIPHER_CTX_block_size()* return the block size.

EVP_CIPHER_key_length() and *EVP_CIPHER_CTX_key_length()* return the key length.

EVP_CIPHER_CTX_set_padding() always returns 1.

EVP_CIPHER_iv_length() and *EVP_CIPHER_CTX_iv_length()* return the IV length or zero if the cipher does not use an IV.

EVP_CIPHER_type() and *EVP_CIPHER_CTX_type()* return the NID of the cipher's OBJECT IDENTIFIER or NID_undef if it has no defined OBJECT IDENTIFIER.

EVP_CIPHER_CTX_cipher() returns an **EVP_CIPHER** structure.

EVP_CIPHER_param_to_asn1() and *EVP_CIPHER_asn1_to_param()* return 1 for success or zero

for failure.

CIPHER LISTING

All algorithms have a fixed key length unless otherwise stated.

EVP_enc_null()
Null cipher: does nothing.

EVP_des_cbc(void), EVP_des_ecb(void), EVP_des_cfb(void), EVP_des_ofb(void)
DES in CBC, ECB, CFB and OFB modes respectively.

EVP_des_ede_cbc(void), *EVP_des_ede()*, EVP_des_ede_ofb(void), EVP_des_ede_cfb(void)
Two key triple DES in CBC, ECB, CFB and OFB modes respectively.

EVP_des_ede3_cbc(void), *EVP_des_ede3()*, EVP_des_ede3_ofb(void), EVP_des_ede3_cfb(void)
Three key triple DES in CBC, ECB, CFB and OFB modes respectively.

EVP_desx_cbc(void)
DESX algorithm in CBC mode.

EVP_rc4(void)
RC4 stream cipher. This is a variable key length cipher with default key length 128 bits.

EVP_rc4_40(void)
RC4 stream cipher with 40 bit key length. This is obsolete and new code should use *EVP_rc4()* and the *EVP_CIPHER_CTX_set_key_length()* function.

EVP_idea_cbc() EVP_idea_ecb(void), EVP_idea_cfb(void), EVP_idea_ofb(void), EVP_idea_cbc(void)
IDEA encryption algorithm in CBC, ECB, CFB and OFB modes respectively.

EVP_rc2_cbc(void), EVP_rc2_ecb(void), EVP_rc2_cfb(void), EVP_rc2_ofb(void)
RC2 encryption algorithm in CBC, ECB, CFB and OFB modes respectively. This is a variable key length cipher with an additional parameter called "effective key bits" or "effective key length". By default both are set to 128 bits.

EVP_rc2_40_cbc(void), EVP_rc2_64_cbc(void)
RC2 algorithm in CBC mode with a default key length and effective key length of 40 and 64 bits. These are obsolete and new code should use*EVP_r c2_cbc()*, *EVP_CIPHER_CTX_set_key_length()* and *EVP_CIPHER_CTX_ctrl()* to set the key length and effective key length.

EVP_bf_cbc(void), EVP_bf_ecb(void), EVP_bf_cfb(void), EVP_bf_ofb(void);
Blowfish encryption algorithm in CBC, ECB, CFB and OFB modes respectively. This is a variable key length cipher.

EVP_cast5_cbc(void), EVP_cast5_ecb(void), EVP_cast5_cfb(void), EVP_cast5_ofb(void)
CAST encryption algorithm in CBC, ECB, CFB and OFB modes respectively. This is a variable key length cipher.

EVP_rc5_32_12_16_cbc(void), EVP_rc5_32_12_16_ecb(void), EVP_rc5_32_12_16_cfb(void), EVP_rc5_32_12_16_ofb(void)
RC5 encryption algorithm in CBC, ECB, CFB and OFB modes respectively. This is a variable key length cipher with an additional "number of rounds" parameter. By default the key length is set to 128 bits and 12 rounds.

EVP_aes_128_gcm(void), EVP_aes_192_gcm(void), EVP_aes_256_gcm(void)
AES Galois Counter Mode (GCM) for 128, 192 and 256 bit keys respectively. These ciphers require additional control operations to function correctly: see "GCM mode" section below for details.

EVP_aes_128_ccm(void), EVP_aes_192_ccm(void), EVP_aes_256_ccm(void)
AES Counter with CBC-MAC Mode (CCM) for 128, 192 and 256 bit keys respectively. These ciphers require additional control operations to function correctly: see CCM mode section

below for details.

GCM Mode

For GCM mode ciphers the behaviour of the EVP interface is subtly altered and several GCM specific ctrl operations are supported.

To specify any additional authenticated data (AAD) a call to *EVP_CipherUpdate()*, *EVP_EncryptUpdate()* or *EVP_DecryptUpdate()* should be made with the output parameter **out** set to **NULL**.

When decrypting the return value of *EVP_DecryptFinal()* or *EVP_CipherFinal()* indicates if the operation was successful. If it does not indicate success the authentication operation has failed and any output data **MUST NOT** be used as it is corrupted.

The following ctrls are supported in GCM mode:

```
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_IVLEN, ivlen, NULL);
```

Sets the GCM IV length: this call can only be made before specifying an IV. If not called a default IV length is used (96 bits for AES).

```
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_GET_TAG, taglen, tag);
```

Writes **taglen** bytes of the tag value to the buffer indicated by **tag**. This call can only be made when encrypting data and **after** all data has been processed (e.g. after an *EVP_EncryptFinal()* call).

```
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_TAG, taglen, tag);
```

Sets the expected tag to **taglen** bytes from **tag**. This call is only legal when decrypting data and must be made **before** any data is processed (e.g. before any *EVP_DecryptUpdate()* call).

See EXAMPLES below for an example of the use of GCM mode.

CCM Mode

The behaviour of CCM mode ciphers is similar to CCM mode but with a few additional requirements and different ctrl values.

Like GCM mode any additional authenticated data (AAD) is passed by calling *EVP_CipherUpdate()*, *EVP_EncryptUpdate()* or *EVP_DecryptUpdate()* with the output parameter **out** set to **NULL**. Additionally the total plaintext or ciphertext length **MUST** be passed to *EVP_CipherUpdate()*, *EVP_EncryptUpdate()* or *EVP_DecryptUpdate()* with the output and input parameters (**in** and **out**) set to **NULL** and the length passed in the **inl** parameter.

The following ctrls are supported in CCM mode:

```
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_CCM_SET_TAG, taglen, tag);
```

This call is made to set the expected **CCM** tag value when decrypting or the length of the tag (with the **tag** parameter set to NULL) when encrypting. The tag length is often referred to as **M**. If not set a default value is used (12 for AES).

```
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_CCM_SET_L, ivlen, NULL);
```

Sets the CCM **L** value. If not set a default is used (8 for AES).

```
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_CCM_SET_IVLEN, ivlen, NULL);
```

Sets the CCM nonce (IV) length: this call can only be made before specifying an nonce value. The nonce length is given by **15 - L** so it is 7 by default for AES.

NOTES

Where possible the **EVP** interface to symmetric ciphers should be used in preference to the low level interfaces. This is because the code then becomes transparent to the cipher used and much more flexible. Additionally, the **EVP** interface will ensure the use of platform specific cryptographic acceleration such as AES-NI (the low level interfaces do not provide the guarantee).

PKCS padding works by adding **n** padding bytes of value **n** to make the total length of the

encrypted data a multiple of the block size. Padding is always added so if the data is already a multiple of the block size **n** will equal the block size. For example if the block size is 8 and 11 bytes are to be encrypted then 5 padding bytes of value 5 will be added.

When decrypting the final block is checked to see if it has the correct form.

Although the decryption operation can produce an error if padding is enabled, it is not a strong test that the input data or key is correct. A random block has better than 1 in 256 chance of being of the correct format and problems with the input data earlier on will not produce a final decrypt error.

If padding is disabled then the decryption operation will always succeed if the total amount of data decrypted is a multiple of the block size.

The functions *EVP_EncryptInit()*, *EVP_EncryptFinal()*, *EVP_DecryptInit()*, *EVP_CipherInit()* and *EVP_CipherFinal()* are obsolete but are retained for compatibility with existing code. New code should use *EVP_EncryptInit_ex()*, *EVP_EncryptFinal_ex()*, *EVP_DecryptInit_ex()*, *EVP_DecryptFinal_ex()*, *EVP_CipherInit_ex()* and *EVP_CipherFinal_ex()* because they can reuse an existing context without allocating and freeing it up on each call.

BUGS

For RC5 the number of rounds can currently only be set to 8, 12 or 16. This is a limitation of the current RC5 code rather than the EVP interface.

EVP_MAX_KEY_LENGTH and EVP_MAX_IV_LENGTH only refer to the internal ciphers with default key lengths. If custom ciphers exceed these values the results are unpredictable. This is because it has become standard practice to define a generic key as a fixed unsigned char array containing EVP_MAX_KEY_LENGTH bytes.

The ASN1 code is incomplete (and sometimes inaccurate) it has only been tested for certain common S/MIME ciphers (RC2, DES, triple DES) in CBC mode.

EXAMPLES

Encrypt a string using IDEA:

```
int do_crypt(char *outfile)
{
unsigned char outbuf[1024];
int outlen, tmplen;
/* Bogus key and IV: we'd normally set these from
* another source.
*/
unsigned char key[] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
unsigned char iv[] = {1,2,3,4,5,6,7,8};
char intext[] = "Some Crypto Text";
EVP_CIPHER_CTX ctx;
FILE *out;

EVP_CIPHER_CTX_init(&ctx);
EVP_EncryptInit_ex(&ctx, EVP_idea_cbc(), NULL, key, iv);

if(!EVP_EncryptUpdate(&ctx, outbuf, &outlen, intext, strlen(intext)))
{
/* Error */
return 0;
}
/* Buffer passed to EVP_EncryptFinal() must be after data just
* encrypted to avoid overwriting it.
*/
if(!EVP_EncryptFinal_ex(&ctx, outbuf + outlen, &tmplen))
```

```
{
/* Error */
return 0;
}
outlen += tmplen;
EVP_CIPHER_CTX_cleanup(&ctx);
/* Need binary mode for fopen because encrypted data is
* binary data. Also cannot use strlen() on it because
* it wont be null terminated and may contain embedded
* nulls.
*/
out = fopen(outfile, "wb");
fwrite(outbuf, 1, outlen, out);
fclose(out);
return 1;
}
```

The ciphertext from the above example can be decrypted using the **openssl** utility with the command line (shown on two lines for clarity):

```
openssl idea -d <filename
-K 000102030405060708090A0B0C0D0E0F -iv 0102030405060708
```

General encryption and decryption function example using FILE I/O and AES128 with a 128-bit key:

```
int do_crypt(FILE *in, FILE *out, int do_encrypt)
{
/* Allow enough space in output buffer for additional block */
unsigned char inbuf[1024], outbuf[1024 + EVP_MAX_BLOCK_LENGTH];
int inlen, outlen;
EVP_CIPHER_CTX ctx;
/* Bogus key and IV: we'd normally set these from
* another source.
*/
unsigned char key[] = "0123456789abcdeF";
unsigned char iv[] = "1234567887654321";

/* Don't set key or IV right away; we want to check lengths */
EVP_CIPHER_CTX_init(&ctx);
EVP_CipherInit_ex(&ctx, EVP_aes_128_cbc(), NULL, NULL, NULL,
do_encrypt);
OPENSSL_assert(EVP_CIPHER_CTX_key_length(&ctx) == 16);
OPENSSL_assert(EVP_CIPHER_CTX_iv_length(&ctx) == 16);

/* Now we can set key and IV */
EVP_CipherInit_ex(&ctx, NULL, NULL, key, iv, do_encrypt);

for(;;)
{
inlen = fread(inbuf, 1, 1024, in);
if(inlen <= 0) break;
if(!EVP_CipherUpdate(&ctx, outbuf, &outlen, inbuf, inlen))
{
/* Error */
EVP_CIPHER_CTX_cleanup(&ctx);
return 0;
```

```
}
fwrite(outbuf, 1, outlen, out);
}
if(!EVP_CipherFinal_ex(&ctx, outbuf, &outlen))
{
/* Error */
EVP_CIPHER_CTX_cleanup(&ctx);
return 0;
}
fwrite(outbuf, 1, outlen, out);

EVP_CIPHER_CTX_cleanup(&ctx);
return 1;
}
```

SEE ALSO

evp(3)

HISTORY

EVP_CIPHER_CTX_init(), *EVP_EncryptInit_ex()*, *EVP_EncryptFinal_ex()*, *EVP_DecryptInit_ex()*, *EVP_DecryptFinal_ex()*, *EVP_CipherInit_ex()*, *EVP_CipherFinal_ex()* and *EVP_CIPHER_CTX_set_padding()* appeared in OpenSSL 0.9.7.

IDEA appeared in OpenSSL 0.9.7 but was often disabled due to patent concerns; the last patents expired in 2012.