NAME

EVP_ENCODE_CTX_new, EVP_ENCODE_CTX_free, EVP_ENCODE_CTX_copy, EVP_ENCODE_CTX_num, EVP_EncodeInit, EVP_EncodeUpdate, EVP_EncodeFinal, EVP_EncodeBlock, EVP_DecodeInit, EVP_DecodeUpdate, EVP_DecodeFinal, EVP_DecodeBlock - EVP base 64 encode/decode routines

SYNOPSIS

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

EVP_ENCODE_CTX *EVP_ENCODE_CTX_new(void);
void EVP_ENCODE_CTX_free(EVP_ENCODE_CTX *ctx);
int EVP_ENCODE_CTX_copy(EVP_ENCODE_CTX *dctx, EVP_ENCODE_CTX *sctx);
int EVP_ENCODE_CTX_num(EVP_ENCODE_CTX *ctx);
void EVP_EncodeInit(EVP_ENCODE_CTX *ctx);
int EVP_EncodeUpdate(EVP_ENCODE_CTX *ctx, unsigned char *out, int *outl,
                     const unsigned char *in, int inl);
void EVP_EncodeFinal(EVP_ENCODE_CTX *ctx, unsigned char *out, int *outl);
int EVP_EncodeBlock(unsigned char *t, const unsigned char *f, int n);

void EVP_DecodeInit(EVP_ENCODE_CTX *ctx);
int EVP_DecodeUpdate(EVP_ENCODE_CTX *ctx, unsigned char *out, int *outl,
                     const unsigned char *in, int inl);
int EVP_DecodeFinal(EVP_ENCODE_CTX *ctx, unsigned char *out, int *outl);
int EVP_DecodeBlock(unsigned char *t, const unsigned char *f, int n);
```

DESCRIPTION

The EVP encode routines provide a high-level interface to base 64 encoding and decoding. Base 64 encoding converts binary data into a printable form that uses the characters A-Z, a-z, 0-9, "+" and "/" to represent the data. For every 3 bytes of binary data provided 4 bytes of base 64 encoded data will be produced plus some occasional newlines (see below). If the input data length is not a multiple of 3 then the output data will be padded at the end using the "=" character.

EVP_ENCODE_CTX_new() allocates, initializes and returns a context to be used for the encode/decode functions.

EVP_ENCODE_CTX_free() cleans up an encode/decode context **ctx** and frees up the space allocated to it.

Encoding of binary data is performed in blocks of 48 input bytes (or less for the final block). For each 48 byte input block encoded 64 bytes of base 64 data is output plus an additional newline character (i.e.

65 bytes in total). The final block (which may be less than 48 bytes) will output 4 bytes for every 3 bytes of input. If the data length is not divisible by 3 then a full 4 bytes is still output for the final 1 or 2 bytes of input. Similarly a newline character will also be output.

EVP_EncodeInit() initialises **ctx** for the start of a new encoding operation.

EVP_EncodeUpdate() encode **inl** bytes of data found in the buffer pointed to by **in**. The output is stored in the buffer **out** and the number of bytes output is stored in ***outl**. It is the caller's responsibility to ensure that the buffer at **out** is sufficiently large to accommodate the output data. Only full blocks of data (48 bytes) will be immediately processed and output by this function. Any remainder is held in the **ctx** object and will be processed by a subsequent call to **EVP_EncodeUpdate()** or **EVP_EncodeFinal()**. To calculate the required size of the output buffer add together the value of **inl** with the amount of unprocessed data held in **ctx** and divide the result by 48 (ignore any remainder). This gives the number of blocks of data that will be processed. Ensure the output buffer contains 65 bytes of storage for each block, plus an additional byte for a NUL terminator. **EVP_EncodeUpdate()** may be called repeatedly to process large amounts of input data. In the event of an error **EVP_EncodeUpdate()** will set ***outl** to 0 and return 0. On success 1 will be returned.

EVP_EncodeFinal() must be called at the end of an encoding operation. It will process any partial block of data remaining in the **ctx** object. The output data will be stored in **out** and the length of the data written will be stored in ***outl**. It is the caller's responsibility to ensure that **out** is sufficiently large to accommodate the output data which will never be more than 65 bytes plus an additional NUL terminator (i.e. 66 bytes in total).

EVP_ENCODE_CTX_copy() can be used to copy a context **sctx** to a context **dctx**. **dctx** must be initialized before calling this function.

EVP_ENCODE_CTX_num() will return the number of as yet unprocessed bytes still to be encoded or decoded that are pending in the **ctx** object.

EVP_EncodeBlock() encodes a full block of input data in **f** and of length **n** and stores it in **t**. For every 3 bytes of input provided 4 bytes of output data will be produced. If **n** is not divisible by 3 then the block is encoded as a final block of data and the output is padded such that it is always divisible by 4. Additionally a NUL terminator character will be added. For example if 16 bytes of input data is provided then 24 bytes of encoded data is created plus 1 byte for a NUL terminator (i.e. 25 bytes in total). The length of the data generated *without* the NUL terminator is returned from the function.

EVP_DecodeInit() initialises **ctx** for the start of a new decoding operation.

EVP_DecodeUpdate() decodes **inl** characters of data found in the buffer pointed to by **in**. The output is

stored in the buffer **out** and the number of bytes output is stored in ***outl**. It is the caller's responsibility to ensure that the buffer at **out** is sufficiently large to accommodate the output data. This function will attempt to decode as much data as possible in 4 byte chunks. Any whitespace, newline or carriage return characters are ignored. Any partial chunk of unprocessed data (1, 2 or 3 bytes) that remains at the end will be held in the **ctx** object and processed by a subsequent call to **EVP_DecodeUpdate()**. If any illegal base 64 characters are encountered or if the base 64 padding character "=" is encountered in the middle of the data then the function returns -1 to indicate an error. A return value of 0 or 1 indicates successful processing of the data. A return value of 0 additionally indicates that the last input data characters processed included the base 64 padding character "=" and therefore no more non-padding character data is expected to be processed. For every 4 valid base 64 bytes processed (ignoring whitespace, carriage returns and line feeds), 3 bytes of binary output data will be produced (or less at the end of the data where the padding character "=" has been used).

EVP_DecodeFinal() must be called at the end of a decoding operation. If there is any unprocessed data still in **ctx** then the input data must not have been a multiple of 4 and therefore an error has occurred. The function will return -1 in this case. Otherwise the function returns 1 on success.

EVP_DecodeBlock() will decode the block of **n** characters of base 64 data contained in **f** and store the result in **t**. Any leading whitespace will be trimmed as will any trailing whitespace, newlines, carriage returns or EOF characters. After such trimming the length of the data in **f** must be divisible by 4. For every 4 input bytes exactly 3 output bytes will be produced. The output will be padded with 0 bits if necessary to ensure that the output is always 3 bytes for every 4 input bytes. This function will return the length of the data decoded or -1 on error.

RETURN VALUES

EVP_ENCODE_CTX_new() returns a pointer to the newly allocated EVP_ENCODE_CTX object or NULL on error.

EVP_ENCODE_CTX_num() returns the number of bytes pending encoding or decoding in **ctx**.

EVP_EncodeUpdate() returns 0 on error or 1 on success.

EVP_EncodeBlock() returns the number of bytes encoded excluding the NUL terminator.

EVP_DecodeUpdate() returns -1 on error and 0 or 1 on success. If 0 is returned then no more non-padding base 64 characters are expected.

EVP_DecodeFinal() returns -1 on error or 1 on success.

EVP_DecodeBlock() returns the length of the data decoded or -1 on error.

SEE ALSO

evp(7)

COPYRIGHT