InterNiche API Reference

Interniche Legacy Document

Version 1.00

Date: 12-May-2017 10:36

Table of Contents

1 CryptoEngine

- ce_dsa_sig_set_rs()
- ce_dsa_sign()
- ce_dsa_size()
- ce_dsa_vrfy()
- ce_hmac_auth()
- ce_hmac_auth_final()
- ce_hmac_auth_init()
- ce_hmac_auth_update()
- ce_rand_bytes()
- ce_rand_cleanup()
- ce_rand_init()
- ce_rand_seed()
- ce_rand_status()
- ce_rsa_d2i_prvkey()
- ce_rsa_d2i_pubkey()
- ce_rsa_d2i_pubkey2()
- ce_rsa_free()
- ce_rsa_get_modulus()
- ce_rsa_get_pub_exp()
- ce_rsa_new()
- ce_rsa_new0()
- ce_rsa_parm_size()
- ce_rsa_prv_decr()
- ce_rsa_prv_encr()
- ce_rsa_pub_decr()
- ce_rsa_pub_encr()
- ce_rsa_rd_prv_key_file()
- ce_rsa_set_modulus_and_pub_exp()
- ce_rsa_sign()
- ce_rsa_size()
- ce_rsa_vrfy()
- ce_sym_priv()
- ce_utils_des_set_odd_parity()
- ce_utils_get_auth_params()
- ce_x509cert_db_add()
- ce_x509cert_db_create()
- ce_x509cert_db_del()
- ce_x509cert_db_destroy()
- ce_x509cert_db_find()
- ce_x509cert_db_list()
- ce_x509cert_free()
- ce_x509cert_get_pubkey()
- ce_x509cert_get_subject_altname()
- ce_x509cert_getparm()
- ce_x509cert_print()
- ce_x509cert_rd_buf()

- ce_x509cert_rd_buf2()
- ce_x509cert_rd_file()
- ce_x509cert_rd_prvkey2()
- ce_x509cert_store_add_trusted_cert()
- ce_x509cert_store_add_untrusted_cert()
- ce_x509cert_store_create()
- ce_x509cert_store_destroy()
- ce_x509cert_store_verify()
- ce_x509cert_verify()
- ce_x509cert_wr_buf()
- ce_x509cert_wr_file()

1.1 ce_auth

API Name

`ce_auth()`

Syntax

```
int ce_auth(CE_OP_PTR op, u_char *in[], int inlen[], int count, u_char
*out, int outlen)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
in	Array of input buffers
inlen	Array of input buffer lengths
count	Number of elements in array
out	Output buffer
outlen	Length of output buffer

Description

This function computes a digest of the specified input buffer(s) using the digest algorithm specified in the cipher identifier.

Returns

This function returns ESUCCESS if the digest is computed successfully; otherwise, it returns EFAILURE.

1.2 ce_auth_final

API Name

`ce_auth_final()`

Syntax

```
int ce_auth_final(CE_OP_PTR op, void *ctx, u_char *out, int outlen)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
ctx	Pointer to plain digest authentication context structure
out	Pointer to output buffer
outlen	Length of output buffer (bytes)

Description

This function finalizes the plain digest computation process, and copies the computed digest into the buffer provided by the caller.

Returns

This function returns ESUCCESS if the requested operation was successful; otherwise it returns EFAILURE.

1.3 ce_auth_init

API Name

`ce_auth_init()`

Syntax

```
void *ce_auth_init(CE_OP_PTR op)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)

Description

This function creates a plain digest authentication context for the digest algorithm specified in the cipher identifier. The digest context is required for all subsequent (incremental) digest computations.

Returns

This function returns a pointer to the digest context, or NULL in the event of an error.

1.4 ce_auth_update

API Name

`ce_auth_update()`

Syntax

```
int ce_auth_update(CE_OP_PTR op, void *ctx, u_char *in, int inlen)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
ctx	Pointer to plain digest authentication context structure
in	Pointer to input buffer containing data to be digested
inlen	Length of input buffer (bytes)

Description

This function feeds the input data provided by the caller into the digest computation algorithm. The intermediate results are stored in the digest context.

Returns

This function returns ESUCCESS if the requested operation was successful; otherwise it returns EFAILURE.

1.5 ce_b64_dec

API Name

`ce_b64_dec()`

Syntax

```
int ce_b64_dec(CE_OP_PTR op, uint8_t *in, int inlen, uint8_t *out, int
*outlen)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
in	Pointer to input buffer containing base64-encoded data
inlen	Length of input buffer
out	Pointer to output buffer
outlen	Pointer to integer containing length of output buffer (input), and pointer to integer to store length of decoded output (output)

Description

This function performs base-64 decoding on the data in the input buffer, and stores the decoded data into the caller-provided output buffer. The length of the decoded data is copied into '*outlen'.

Returns

This function returns ESUCCESS if the requested operation was completed successfully; otherwise, it returns EFAILURE.

1.6 ce_b64_enc

API Name

`ce_b64_enc()`

Syntax

```
int ce_b64_enc(CE_OP_PTR op, uint8_t *in, int inlen, uint8_t *out, int outlen)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
in	Pointer to input buffer
inlen	Length of input buffer
out	Pointer to output buffer
outlen	Length of output buffer

Description

This function performs base-64 encoding on the data in the input buffer, and stores the encoded data into the caller-provided output buffer.

Returns

This function returns ESUCCESS if the requested operation was completed successfully; otherwise, it returns EFAILURE.

1.7 ce_bn_bin2bn

API Name

`ce_bn_bin2bn()`

Syntax

```
void *ce_bn_bin2bn(CE_OP_PTR op, u_char *in, int inlen, void *bn);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
in	Pointer to array of bytes comprising the input bignum (in big-endian order)
inlen	Length of the array of bytes comprising the input bignum
bn	Pointer to output bignum structure that will be filled with the big number stored at 'in'

Description

This function "fills" the bignum structure at 'bn' with the value of the big number at 'in'.

Returns

Returns a pointer to the filled-in bignum structure ('bn'), or NULL in the event of an error.

1.8 ce_bn_bn2bin

API Name

`ce_bn_bn2bin()`

Syntax

```
int ce_bn_bn2bin(CE_OP_PTR op, void *bn, u_char *out);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
bn	Pointer to input bignum structure
out	Pointer to array of bytes where the bignum will be stored (in big-endian order)

Description

This function copies out the bignum stored at 'bn' into 'out'. The caller must ensure that 'out' points to storage of adequate size.

Returns

Returns ESUCCESS if the bignum was written out successfully, and EFAILURE in the event of an error.

1.9 ce_bn_dup

API Name

`ce_bn_dup()`

Syntax

```
void *ce_bn_dup(CE_OP_PTR op, void *bn);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
bn	Pointer to structure containing bignum that is to be copied

Description

This function creates a copy of the bignum stored at 'bn'.

Returns

Returns a pointer to the duplicate bignum structure, or NULL in the event that a duplicate could not be created.

1.10 ce_bn_free

API Name

`ce_bn_free()`

Syntax

```
void ce_bn_free(CE_OP_PTR op, void *bn);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
bn	Pointer to structure containing bignum

Description

This function frees the bignum structure located at 'bn'.

Returns

None.

1.11 ce_bn_hex2bn

API Name

`ce_bn_hex2bn()`

Syntax

```
void *ce_bn_hex2bn(CE_OP_PTR op, void *bn, char *in);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
bn	Pointer to pointer to output bignum structure that will be filled with the big number stored at 'in'
in	Pointer to (NULL-terminated) hexadecimal string containing the input bignum (in big-endian order)

Description

This function "fills" the bignum structure at '*bn' with the value of the big number at 'in'. If '*bn' is NULL, a new bignum structure is allocated prior to use.

Returns

This function returns ESUCCESS if the input data was copied into the bignum structure, and EFAILURE in the event of an error.

1.12 ce_bn_new

API Name

`ce_bn_new()`

Syntax

```
void *ce_bn_new(CE_OP_PTR op);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)

Description

This function allocates a new bignum structure.

Returns

Pointer to bignum structure, or NULL if the structure could not be allocated.

1.13 ce_bn_num_bytes

API Name

`ce_bn_num_bytes()`

Syntax

```
int ce_bn_num_bytes(CE_OP_PTR op, void *bn);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
bn	Pointer to structure containing bignum

Description

This function returns the size (in bytes) of the bignum stored at 'bn'.

Returns

Size of big number stored at 'bn'.

1.14 ce_dh_free

API Name

`ce_dh_free()`

Syntax

```
void ce_dh_free(CE_OP_PTR op, void *dh);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
dh	Pointer to Diffie-Hellman (DH) structure

Description

This function frees a DH structure.

Returns

None.

1.15 ce_dh_gen_local_pubpriv

API Name

`ce_dh_gen_local_pubpriv()`

Syntax

```
int ce_dh_gen_local_pubpriv(CE_OP_PTR op, void *dh, u_char *lcl_pub, int
*pub_len, u_char *lcl_prv, int *prv_len);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
dh	Pointer to Diffie-Hellman (DH) structure
lcl_pub	Output buffer to store public DH value
pub_len	Length of 'lcl_pub' buffer
lcl_prv	Output buffer to store private DH value
prv_len	Length of 'lcl_prv' buffer

Description

This function generates the public ('g^x_mod(p)') and private ('x') DH values.

Returns

This function returns ESUCCESS if the generation of public and private DH values is successful; otherwise, it returns EFAILURE. If an output buffer is specified as NULL, or if the buffer is insufficiently long, this function will write the length of the corresponding parameter to '*pub_len' (for the public DH value) or '*prv_len' (for the private DH value).

1.16 ce_dh_gen_shared_secret

API Name

`ce_dh_gen_shared_secret()`

Syntax

```
int ce_dh_gen_shared_secret(CE_OP_PTR op, void *dh, u_char *secret, int
*secret_len, const u_char *rem_pub, int rem_len);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
dh	Pointer to Diffie-Hellman (DH) structure
secret	Output buffer (for storing DH shared secret)
secret_len	Length of output buffer (input) and required length (output)
rem_pub	Public DH value of remote end (peer)
rem_len	Length of peer's public DH value

Description

This function computes the DH shared secret, and copies it into the specified output buffer. The length of the computed secret is copied into '*secret_len'.

Returns

This function returns ESUCCESS if the shared secret computation is successful; otherwise, it returns EFAILURE.

1.17 ce_dh_gen_shared_secret0

API Name

`ce_dh_gen_shared_secret0()`

Syntax

```
int ce_dh_gen_shared_secret0(CE_OP_PTR op, void *dh, u_char *secret, void
*rem_pub);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
dh	Pointer to Diffie-Hellman (DH) structure
secret	Output buffer (for storing DH shared secret)
rem_pub	Pointer to big number containing public DH value of remote end (peer)

Description

This function computes the DH shared secret, and copies it into the specified output buffer.

Returns

This function returns ESUCCESS if the shared secret computation is successful; otherwise, it returns EFAILURE.

1.18 ce_dh_gen_shared_secret2

API Name

`ce_dh_gen_shared_secret2()`

Syntax

```
int ce_dh_gen_shared_secret2(CE_OP_PTR op, u_char *rem_pub, int rem_len,
u_char *lcl_priv, int lcl_len, u_char *secret, int *secret_len);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
rem_pub	Pointer to input buffer containing public DH value of remote end (peer)
rem_len	Length of peer's public DH value
lcl_priv	Pointer to input buffer containing private DH value of local end
lcl_len	Length of local end's private DH value
secret	Output buffer (for storing DH shared secret)
secret_len	Pointer to integer containing length of output buffer (input) and required length (output)

Description

This function computes the DH shared secret, and copies it into the specified output buffer. The length of the computed secret is copied into '*secret_len'.

Returns

This function returns ESUCCESS if the shared secret computation is successful; otherwise, it returns EFAILURE.

1.19 ce_dh_get_pub

API Name

`ce_dh_get_pub()`

Syntax

```
void *ce_dh_get_pub(CE_OP_PTR op, void *dh);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
dsa	Pointer to Diffie-Hellman structure

Description

This function returns a pointer to the big number that comprises the public value for the specified Diffie-Hellman key structure.

Returns

Pointer to the big number that comprises the public value for the specified Diffie-Hellman key structure.

1.20 ce_dh_new

API Name

`ce_dh_new()`

Syntax

```
void *ce_dh_new(CE_OP_PTR op, bool_t init, int len, int generator);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
init	Boolean that determines whether an "empty" (0) or filled-in (1) Diffie-Hellman (DH) structure is returned
len	Length in bits of safe prime ('p') to be generated
generator	Generator ('g', 2 or 5)

Description

This function allocates a new (empty or initialized) Diffie-Hellman (DH) structure.

Returns

Pointer to DH structure, or NULL in the event of an error.

1.21 ce_dh_setparms_pg

API Name

`ce_dh_setparms_pg()`

Syntax

```
int ce_dh_setparms_pg(CE_OP_PTR op, void *dh, const u_char *p, int plen,
const u_char *g, int glen);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
dh	Pointer to Diffie-Hellman (DH) structure
p	Pointer to input buffer containing prime ('p' in g^x_mod(p))
plen	Length of prime (in bytes)
g	Pointer to input buffer containing generator ('g' in g^x_mod(p))
glen	Length of generator (in bytes)

Description

This function configures the prime 'p' and generator 'g' in the Diffie-Hellman structure. The 'p' ang 'g' parameters must point to an array of bytes in network byte order that comprise the prime and generator respectively.

Returns

This function returns ESUCCESS if the Diffie-Hellman structure was successfully configured; otherwise, it returns EFAILURE.

1.22 ce_dh_setparms_pg2

API Name

`ce_dh_setparms_pg2()`

Syntax

```
int ce_dh_setparms_pg2(CE_OP_PTR op, void *dh, void *p, void *g);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
dh	Pointer to Diffie-Hellman (DH) structure
p	Pointer to big number containing prime ('p' in g^x_mod(p))
g	Pointer to big number containing generator ('g' in g^x_mod(p))

Description

This function configures the prime 'p' and generator 'g' in the Diffie-Hellman structure.

Returns

This function returns ESUCCESS if the Diffie-Hellman structure was successfully configured; otherwise, it returns EFAILURE.

1.23 ce_dh_size

API Name

`ce_dh_size()`

Syntax

```
int ce_dh_size(CE_OP_PTR op, void *dh);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
dh	Pointer to Diffie-Hellman (DH) structure

Description

This function returns the size (in bytes) of the Diffie-Hellman prime ('p').

Returns

Size (in bytes) of the Diffie-Hellman prime ('p').

1.24 ce_dsa_d2i_prvkey

API Name

`ce_dsa_d2i_prvkey()`

Syntax

```
void *ce_dsa_d2i_prvkey(CE_OP_PTR op, u_char *buf, int len);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
buf	Pointer to input buffer containing DER-encoded DSA private key
len	Length of input buffer

Description

This function returns a pointer to a DSA key structure that is created from the DER-encoded DSA private key.

Returns

Pointer to DSA key structure, or NULL in the event of an error.

1.25 ce_dsa_d2i_pubkey

API Name

`ce_dsa_d2i_pubkey()`

Syntax

```
void *ce_dsa_d2i_pubkey(CE_OP_PTR op, u_char *buf, int len);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
buf	Pointer to input buffer containing DER-encoded DSA public key
len	Length of input buffer

Description

This function returns a pointer to a DSA key structure that is created from the DER-encoded DSA public key.

Returns

Pointer to DSA key structure, or NULL in the event of an error.

1.26 ce_dsa_d2i_pubkey2

API Name

`ce_dsa_d2i_pubkey2()`

Syntax

```
void *ce_dsa_d2i_pubkey2(CE_OP_PTR op, u_char *buf, int len);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
buf	Pointer to input buffer containing DER-encoded DSA public key (in the SubjectPublicKeyInfo format)
len	Length of input buffer

Description

This function returns a pointer to a DSA key structure that is created from the DER-encoded DSA public key (in the SubjectPublicKeyInfo format). This format is described in RFC 2459 ("Internet X.509 Public Key Infrastructure").

Returns

Pointer to DSA key structure, or NULL in the event of an error.

1.27 ce_dsa_d2i_sig

API Name

`ce_dsa_d2i_sig()`

Syntax

```
void *ce_dsa_d2i_sig(CE_OP_PTR op, u_char *buf, int len);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
buf	Pointer to input buffer containing DER-encoded DSA signature
len	Length of input buffer

Description

This function returns a pointer to a DSA signature structure that is created from the DER-encoded DSA signature.

Returns

Pointer to DSA signature structure, or NULL in the event of an error.

1.28 ce_dsa_free

API Name

`ce_dsa_free()`

Syntax

```
void ce_dsa_free(CE_OP_PTR op, void *dsa);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
dsa	Pointer to DSA structure

Description

This function frees a DSA structure.

Returns

None.

1.29 ce_dsa_get_g

API Name

`ce_dsa_get_g()`

Syntax

```
void *ce_dsa_get_g(CE_OP_PTR op, void *dsa);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
dsa	Pointer to DSA key structure

Description

This function returns a pointer to the big number that comprises 'g' for the specified DSA key structure. DSA parameters are described in FIPS PUB 186-4 ("Digital Signature Standard").

Returns

Pointer to the big number that comprises 'g' for the specified DSA key structure.

1.30 ce_dsa_get_p

API Name

`ce_dsa_get_p()`

Syntax

```
void *ce_dsa_get_p(CE_OP_PTR op, void *dsa);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
dsa	Pointer to DSA key structure

Description

This function returns a pointer to the big number that comprises 'p' for the specified DSA key structure. DSA parameters are described in FIPS PUB 186-4 ("Digital Signature Standard").

Returns

Pointer to the big number that comprises 'p' for the specified DSA key structure.

1.31 ce_dsa_get_pub_key

API Name

`ce_dsa_get_pub_key()`

Syntax

```
void *ce_dsa_get_pub_key(CE_OP_PTR op, void *dsa);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
dsa	Pointer to DSA key structure

Description

This function returns a pointer to the big number that comprises 'y' (public key) for the specified DSA key structure. DSA parameters are described in FIPS PUB 186-4 ("Digital Signature Standard").

Returns

Pointer to the big number that comprises 'y' (public key) for the specified DSA key structure.

1.32 ce_dsa_get_q

API Name

`ce_dsa_get_q()`

Syntax

```
void *ce_dsa_get_q(CE_OP_PTR op, void *dsa);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
dsa	Pointer to DSA key structure

Description

This function returns a pointer to the big number that comprises 'q' for the specified DSA key structure. DSA parameters are described in FIPS PUB 186-4 ("Digital Signature Standard").

Returns

Pointer to the big number that comprises 'q' for the specified DSA key structure.

1.33 ce_dsa_i2d_sig

API Name

`ce_dsa_i2d_sig()`

Syntax

```
int ce_dsa_i2d_sig(CE_OP_PTR op, void *sig, u_char *buf, int *len);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
dsa	Pointer to DSA signature structure
buf	Pointer to output buffer
len	Pointer to integer containing length of output buffer (input), and actual number of bytes copied into output buffer (output)

Description

This function writes out a DSA signature into the specified output buffer. If the 'buf' argument is NULL or if the buffer is insufficiently long, this function will write the amount of space required into '*len'.

Returns

This function returns ESUCCESS if the requested operation was successfully completed; otherwise, it returns EFAILURE.

1.34 ce_dsa_new

API Name

`ce_dsa_new()`

Syntax

```
void *ce_dsa_new(CE_OP_PTR op, int group_size, int modulus_size);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
group_size	Group size
modulus_size	Modulus size

Description

This function allocates a new DSA structure. The capabilities provided by this function vary by cryptography provider.

Returns

Pointer to DSA structure, or NULL in the event of a failure.

1.35 ce_dsa_new0

API Name

`ce_dsa_new0()`

Syntax

```
void *ce_dsa_new0(CE_OP_PTR op);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)

Description

This function allocates an "empty" (uninitialized) DSA key structure.

Returns

Pointer to DSA key structure, or NULL in the event of an error.

1.36 ce_dsa_parm_size

API Name

`ce_dsa_parm_size()`

Syntax

```
int ce_dsa_parm_size(CE_OP_PTR op, void *dsa, DSA_PARM parm_id);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
dsa	Pointer to DSA structure
parm_id	Identifier for DSA parameter

Description

This function returns the size (in bytes) of the specified parameter (CE_DSA_PARM_G, CE_DSA_PARM_P, CE_DSA_PARM_Q, CE_DSA_PARM_X (private key), CE_DSA_PARM_Y (public key)) for the specified key.

Returns

Size (in bytes) of requested parameter.

1.37 ce_dsa_rd_prv_key_file

API Name

`ce_dsa_rd_prv_key_file()`

Syntax

```
void *ce_dsa_rd_prv_key_file(CE_OP_PTR op, char *file_name, int file_type,
void *passphrase);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
file_name	Name of file containing DSA private key
file_type	Type of file (unused)
passphrase	Passphrase to decrypt file (required only if private key is encrypted)

Description

This function creates a DSA private key from the contents of the specified file. The private key file must be encoded in the PEM format. If the private key is stored in the clear, the 'passphrase' parameter must be specified as NULL.

Returns

Pointer to DSA key structure.

1.38 ce_dsa_set_pqgy

API Name

`ce_dsa_set_pqgy()`

Syntax

```
int ce_dsa_set_pqgy(CE_OP_PTR op, void *dsa, void *p, void *q, void *g,
void *pub_key);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
dsa	Pointer to DSA key structure
p	Pointer to big number containing 'p'
q	Pointer to big number containing 'q'
g	Pointer to big number containing 'g'
pub_key	Pointer to big number containing 'y' (public key)

Description

This function sets the 'p', 'q', 'g', and 'y' (public key) parameters for the specified DSA key structure to the specified values. DSA parameters are described in FIPS PUB 186-4 ("Digital Signature Standard").

Returns

This function returns ESUCCESS if the requested operation was successful; otherwise, it returns EFAILURE.

1.39 ce_dsa_sig_free

API Name

`ce_dsa_sig_free()`

Syntax

```
void ce_dsa_sig_free(CE_OP_PTR op, void *sig);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
sig	Pointer to DSA signature structure

Description

This function frees a DSA signature structure.

Returns

None.

1.40 ce_dsa_sig_get_r

API Name

`ce_dsa_sig_get_r()`

Syntax

```
void *ce_dsa_sig_get_r(CE_OP_PTR op, void *sig);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
dsa	Pointer to DSA signature structure

Description

This function returns a pointer to the big number that comprises 'r' for the specified DSA signature structure. DSA parameters are described in FIPS PUB 186-4 ("Digital Signature Standard").

Returns

Pointer to the big number that comprises 'r' for the specified DSA signature structure.

1.41 ce_dsa_sig_get_s

API Name

`ce_dsa_sig_get_s()`

Syntax

```
void *ce_dsa_sig_get_s(CE_OP_PTR op, void *sig);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
dsa	Pointer to DSA signature structure

Description

This function returns a pointer to the big number that comprises 's' for the specified DSA signature structure. DSA parameters are described in FIPS PUB 186-4 ("Digital Signature Standard").

Returns

Pointer to the big number that comprises 's' for the specified DSA signature structure.

1.42 ce_dsa_sig_new

API Name

`ce_dsa_sig_new()`

Syntax

```
void *ce_dsa_sig_new(CE_OP_PTR op);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)

Description

This function allocates an "empty" (uninitialized) DSA signature structure.

Returns

Pointer to DSA signature structure, or NULL in the event of an error.

1.43 ce_dsa_sig_set_rs

API Name

`ce_dsa_sig_set_rs()`

Syntax

```
int ce_dsa_sig_set_rs(CE_OP_PTR op, void *sig, void *r, void *s);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
sig	Pointer to DSA signature structure
r	Pointer to big number containing 'r'
s	Pointer to big number containing 's'

Description

This function sets the 'r' and 's' parameters for the specified DSA signature structure to the specified values. DSA parameters are described in FIPS PUB 186-4 ("Digital Signature Standard").

Returns

This function returns ESUCCESS if the requested operation was successful; otherwise, it returns EFAILURE.

1.44 ce_dsa_sign

API Name

`ce_dsa_sign()`

Syntax

```
int ce_dsa_sign(CE_OP_PTR op, int alg, const u_char *in, int inlen, u_char
*out, unsigned int *outlen, void *dsa)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
alg	(unused)
in	Input buffer
inlen	Length of input buffer
out	Output buffer
outlen	Pointer to integer containing length of input buffer (input), and pointer to integer to store length of signature (output)
dsa	Pointer to DSA key structure

Description

This function computes the DSA signature on an input buffer, and copies it into the caller-provided output buffer. The length of the signature is copied into '*outlen'.

Returns

This function returns ESUCCESS if the signature was computed successfully; otherwise, it returns EFAILURE.

1.45 ce_dsa_size

API Name

`ce_dsa_size()`

Syntax

```
int ce_dsa_size(CE_OP_PTR op, void *dsa);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
dsa	Pointer to DSA structure

Description

This function returns the length of an ASN.1-encoded DSA signature (in bytes).

Returns

Modulus size

1.46 ce_dsa_vrfy

API Name

`ce_dsa_vrfy()`

Syntax

```
int ce_dsa_vrfy(CE_OP_PTR op, int alg, const u_char *in, int inlen, u_char
*sig, int siglen, void *dsa)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
alg	(unused)
in	Pointer to input
inlen	Length of input
sig	Pointer to signature
siglen	Length of signature
dsa	Pointer to DSA key structure

Description

This function validates the DSA signature on an input buffer.

Returns

This function returns ESUCCESS if the signature was verified successfully; otherwise, it returns EFAILURE.

1.47 ce_hmac_auth

API Name

`ce_hmac_auth()`

Syntax

```
int ce_hmac_auth(CE_OP_PTR op, u_char *in[], int inlen[], int count, u_char
*key, int keylen, u_char *out, int outlen)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
in	Array of input buffers
inlen	Array of input buffer lengths
count	Number of elements in array
key	Key buffer
keylen	Length of key buffer
out	Output buffer
outlen	Length of output buffer

Description

This function computes the HMAC digest of the specified input buffer(s) using the digest algorithm specified in the cipher identifier.

Returns

This function returns ESUCCESS if the digest is computed successfully; otherwise, it returns EFAILURE.

1.48 ce_hmac_auth_final

API Name

`ce_hmac_auth_final()`

Syntax

```
int ce_hmac_auth_final(CE_OP_PTR op, void *ctx, u_char *out, int outlen)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
ctx	Pointer to HMAC digest authentication context structure
out	Pointer to output buffer
outlen	Length of output buffer (bytes)

Description

This function finalizes the HMAC digest computation process, and copies the computed digest into the buffer provided by the caller.

Returns

This function returns ESUCCESS if the requested operation was successful; otherwise it returns EFAILURE.

1.49 ce_hmac_auth_init

API Name

`ce_hmac_auth_init()`

Syntax

```
void *ce_hmac_auth_init(CE_OP_PTR op, u_char *key, int keylen)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
key	Pointer to input buffer containing key
keylen	Length of key (bytes)

Description

This function creates a HMAC digest authentication context for the digest algorithm specified in the cipher identifier. The digest context is required for all subsequent (incremental) digest computations.

Returns

This function returns a pointer to the digest context, or NULL in the event of an error.

1.50 ce_hmac_auth_update

API Name

`ce_hmac_auth_update()`

Syntax

```
int ce_hmac_auth_update(CE_OP_PTR op, void *ctx, u_char *in, int inlen)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
ctx	Pointer to HMAC digest authentication context structure
in	Pointer to input buffer containing data to be digested
inlen	Length of input buffer (bytes)

Description

This function feeds the input data provided by the caller into the digest computation algorithm. The intermediate results are stored in the digest context.

Returns

This function returns ESUCCESS if the requested operation was successful; otherwise it returns EFAILURE.

1.51 ce_rand_bytes

API Name

`ce_rand_bytes()`

Syntax

```
int ce_rand_bytes(CE_OP_PTR op, u_char *buf, int len);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
buf	Output buffer
len	Length of output buffer

Description

This function generates the requested number of random bytes, and copies them into the caller-provided buffer.

Returns

This function returns ESUCCESS if it can provide the requested data; otherwise, it returns EFAILURE.

1.52 ce_rand_cleanup

API Name

`ce_rand_cleanup()`

Syntax

```
int ce_rand_cleanup(CE_OP_PTR op);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)

Description

This function cleans up the internal state of the specified random number generator.

Returns

This function returns ESUCCESS if the requested operation was successfully performed; otherwise, it returns EFAILURE.

1.53 ce_rand_init

API Name

`ce_rand_init()`

Syntax

```
int ce_rand_init(CE_OP_PTR op);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)

Description

This function initializes the specified random number generator.

Returns

This function returns ESUCCESS if the requested operation was completed successfully; otherwise, it returns EFAILURE.

1.54 ce_rand_seed

API Name

`ce_rand_seed()`

Syntax

```
int ce_rand_seed(CE_OP_PTR op, u_char *buf, int len);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
buf	Input buffer
len	Length of input buffer

Description

This function seeds the random number generator with the caller-provided random data.

Returns

This function always returns ESUCCESS.

1.55 ce_rand_status

API Name

`ce_rand_status()`

Syntax

```
int ce_rand_status(CE_OP_PTR op);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)

Description

This function returns the status of the specified random number generator.

Returns

This function returns ESUCCESS if the specified random number generator is up and running; otherwise, it returns EFAILURE.

1.56 ce_rsa_d2i_prvkey

API Name

`ce_rsa_d2i_prvkey()`

Syntax

```
void *ce_rsa_d2i_prvkey(CE_OP_PTR op, u_char *key, int len);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
key	Pointer to input buffer containing RSA private key
len	Length of input buffer

Description

This function returns a pointer to a RSA key structure that is created from the DER-encoded RSA private key. The latter is stored in the PKCS#1 RSAPrivateKey format (RFC 2437, "PKCS #1: RSA Cryptography Specifications").

Returns

Pointer to RSA key structure.

1.57 ce_rsa_d2i_pubkey

API Name

`ce_rsa_d2i_pubkey()`

Syntax

```
void *ce_rsa_d2i_pubkey(CE_OP_PTR op, u_char *key, int len);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
key	Pointer to input buffer containing RSA public key
len	Length of input buffer

Description

This function returns a pointer to a RSA key structure that is created from the DER-encoded RSA public key. The latter is stored in the PKCS#1 RSAPublicKey format (RFC 2437, "PKCS #1: RSA Cryptography Specifications").

Returns

Pointer to RSA key structure.

1.58 ce_rsa_d2i_pubkey2

API Name

`ce_rsa_d2i_pubkey2()`

Syntax

```
void *ce_rsa_d2i_pubkey2(CE_OP_PTR op, u_char *key, int len);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
key	Pointer to input buffer containing RSA public key
len	Length of input buffer

Description

This function returns a pointer to a RSA key structure that is created from the DER-encoded RSA public key. The latter is stored in the SubjectPublicKeyInfo format (RFC 2459, "Internet X.509 Public Key Infrastructure").

Returns

Pointer to RSA key structure.

1.59 ce_rsa_free

API Name

`ce_rsa_free()`

Syntax

```
void ce_rsa_free(CE_OP_PTR op, void *rsa);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
rsa	Pointer to RSA key structure

Description

This function frees a RSA key structure.

Returns

None.

1.60 ce_rsa_get_modulus

API Name

`ce_rsa_get_modulus()`

Syntax

```
void *ce_rsa_get_modulus(CE_OP_PTR op, void *rsa);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
rsa	Pointer to RSA key structure

Description

This function returns a pointer to the big number that comprises the modulus for the specified RSA key structure.

Returns

Pointer to the big number that comprises the modulus for the specified RSA key structure.

1.61 ce_rsa_get_pub_exp

API Name

`ce_rsa_get_pub_exp()`

Syntax

```
void *ce_rsa_get_pub_exp(CE_OP_PTR op, void *rsa);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
rsa	Pointer to RSA key structure

Description

This function returns a pointer to the big number that comprises the public exponent for the specified RSA key structure.

Returns

Pointer to the big number that comprises the public exponent for the specified RSA key structure.

1.62 ce_rsa_new

API Name

`ce_rsa_new()`

Syntax

```
void *ce_rsa_new(CE_OP_PTR op, int size, int e);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
size	Length of modulus (bits)
e	Public exponent

Description

This function allocates a new RSA key structure.

Returns

Pointer to RSA key structure, or NULL in the event of an error.

1.63 ce_rsa_new0

API Name

`ce_rsa_new0()`

Syntax

```
void *ce_rsa_new0(CE_OP_PTR op);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)

Description

This function allocates an "empty" (uninitialized) RSA key structure.

Returns

Pointer to RSA key structure, or NULL in the event of an error.

1.64 ce_rsa_parm_size

API Name

`ce_rsa_parm_size()`

Syntax

```
int ce_rsa_parm_size(CE_OP_PTR op, void *rsa, RSA_PARM parm_id);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
rsa	Pointer to RSA structure
parm_id	Identifier for RSA parameter

Description

This function returns the size (in bytes) of the specified parameter (CE_RSA_PARM_D (private exponent), CE_RSA_PARM_E (public exponent), or CE_RSA_PARM_N (modulus)) for the specified key.

Returns

Size (in bytes) of requested parameter.

1.65 ce_rsa_prv_decr

API Name

`ce_rsa_prv_decr()`

Syntax

```
int ce_rsa_prv_decr(CE_OP_PTR op, u_char *in, int inlen, u_char *out, int
outlen, void *rsa, int padding);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
in	Input buffer
inlen	Length of input buffer
out	Output buffer
outlen	Length of output buffer
rsa	Pointer to RSA key data structure.
padding	Type of padding (CE_RSA_PAD_NONE, CE_RSA_PAD_PKCS1, CE_RSA_PAD_PKCS1_OAEP, or CE_RSA_PAD_SSLV23).

Description

This function decrypts the input buffer using a RSA private key, and stores the output in the caller-provided buffer. The type of padding supported varies by cryptography provider.

Returns

This function returns the length of the output if the decryption was successful; otherwise, it returns EFAILURE.

1.66 ce_rsa_prv_encr

API Name

`ce_rsa_prv_encr()`

Syntax

```
int ce_rsa_prv_encr(u_char *in, int inlen, u_char *out, int outlen, void
*rsa, int padding);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
in	Input buffer
inlen	Length of input buffer
out	Output buffer
outlen	Length of output buffer
rsa	Pointer to RSA key data structure.
padding	Type of padding (CE_RSA_PAD_NONE, CE_RSA_PAD_PKCS1, CE_RSA_PAD_PKCS1_OAEP, or CE_RSA_PAD_SSLV23)

Description

This function encrypts the input buffer using a RSA private key, and stores the output in the caller-provided buffer. The type of padding supported varies by cryptography provider.

Returns

This function returns the length of the output if the encryption was successful; otherwise, it returns EFAILURE.

1.67 ce_rsa_pub_decr

API Name

`ce_rsa_pub_decr()`

Syntax

```
int ce_rsa_pub_decr(u_char *in, int inlen, u_char *out, int outlen, void
*rsa, int padding);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
in	Input buffer
inlen	Length of input buffer
out	Output buffer
outlen	Length of output buffer
rsa	Pointer to RSA key data structure.
padding	Type of padding (CE_RSA_PAD_NONE, CE_RSA_PAD_PKCS1, CE_RSA_PAD_PKCS1_OAEP, or CE_RSA_PAD_SSLV23)

Description

This function decrypts the input buffer using a RSA public key, and stores the output in the caller-provided buffer. The type of padding supported varies by cryptography provider.

Returns

This function returns the length of the output if the decryption was successful; otherwise, it returns EFAILURE.

1.68 ce_rsa_pub_encr

API Name

`ce_rsa_pub_encr()`

Syntax

```
int ce_rsa_pub_encr(u_char *in, int inlen, u_char *out, int outlen, void
*rsa, int padding);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
in	Input buffer
inlen	Length of input buffer
out	Output buffer
outlen	Length of output buffer
rsa	Pointer to RSA key data structure
padding	Type of padding (CE_RSA_PAD_NONE, CE_RSA_PAD_PKCS1, CE_RSA_PAD_PKCS1_OAEP, or CE_RSA_PAD_SSLV23)

Description

This function encrypts the input buffer using a RSA public key, and stores the output in the caller-provided buffer. The type of padding supported varies by cryptography provider.

Returns

This function returns the length of the output if the encryption was successful; otherwise, it returns EFAILURE.

1.69 ce_rsa_rd_prv_key_file

API Name

`ce_rsa_rd_prv_key_file()`

Syntax

```
void *ce_rsa_rd_prv_key_file(CE_OP_PTR op, char *file_name, int file_type,
void *passphrase);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
file_name	Name of file containing RSA private key
file_type	Type of file (unused)
passphrase	Passphrase required to decrypt file (only if private key is encrypted)

Description

This function creates a RSA private key from the contents of the specified file. The private key in the file must be in the PKCS#1 RSAPrivateKey format (RFC 2437, "PKCS #1: RSA Cryptography Specifications"), and the file must be encoded in the PEM format. If the private key is stored in the clear, the 'passphrase' parameter must be specified as NULL.

Returns

Pointer to RSA key structure.

1.70 ce_rsa_set_modulus_and_pub_exp

API Name

`ce_rsa_set_modulus_and_pub_exp()`

Syntax

```
int ce_rsa_set_modulus_and_pub_exp(CE_OP_PTR op, void *rsa, void *modulus,
void *pub_exp);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
rsa	Pointer to RSA key structure
modulus	Pointer to big number containing modulus
pub_exp	Pointer to big number containing public exponent

Description

This function sets the modulus and public exponent for the specified RSA key structure to the specified values.

Returns

This function returns ESUCCESS if the requested operation was successful; otherwise, it returns EFAILURE.

1.71 ce_rsa_sign

API Name

`ce_rsa_sign()`

Syntax

```
int ce_rsa_sign(int alg, u_char *in, unsigned int inlen, u_char *out,
unsigned int *outlen, void *rsa)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
alg	Algorithm used to generate message digest (stored in the input buffer)
in	Input buffer
inlen	Length of input buffer
out	Output buffer
outlen	(output) Pointer to location to store length of signature
rsa	Pointer to RSA key structure

Description

This function computes the RSA signature on an input buffer, and copies it into the caller-provided output buffer. The length of the signature is copied into '*outlen'.

Returns

This function returns ESUCCESS if the signature was computed successfully; otherwise, it returns EFAILURE.

1.72 ce_rsa_size

API Name

`ce_rsa_size()`

Syntax

```
int ce_rsa_size(CE_OP_PTR op, void *rsa);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
rsa	Pointer to RSA structure

Description

This function returns the modulus size in bytes.

Returns

Modulus size.

1.73 ce_rsa_vrfy

API Name

`ce_rsa_vrfy()`

Syntax

```
int ce_rsa_vrfy(int alg, u_char *in, unsigned int inlen, u_char *sig,
unsigned int siglen, void *rsa)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
alg	Algorithm used to generate message digest
in	Pointer to input
inlen	Length of input
sig	Pointer to signature
siglen	Length of signature
rsa	Pointer to RSA key structure

Description

This function validates the RSA signature on an input buffer.

Returns

This function returns ESUCCESS if the signature was verified successfully; otherwise, it returns EFAILURE.

1.74 ce_sym_priv

API Name

`ce_sym_priv()`

Syntax

```
int ce_sym_priv(CE_OP_PTR op, u_char *in, int inlen, u_char *iv, int ivlen,
u_char *key, int keylen, u_char *out, int outlen, void *vptr,
CE_PRIV_BLK_DESC mode)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
in	Input buffer
inlen	Length of input buffer
iv	Initialization Vector (IV) buffer
ivlen	Length of IV
key	Key buffer
keylen	Length of key buffer
out	Output buffer
outlen	Length of output buffer
vptr	Pointer to structure used in CFB, CTR, and OFB modes of operation (used for both input and output)
mode	Specifies mode of requested operation (encryption or decryption, update IV or don't update IV (CBC mode), and number of bits fed back (CFB and OFB modes))

Description

This function encrypts (or decrypts) the input buffer using the privacy algorithm specified in the cipher identifier.

Returns

This function returns ESUCCESS if the encryption (or decryption) is performed successfully; otherwise, it returns EFAILURE.

1.75 ce_utils_des_set_odd_parity

API Name

`ce_utils_des_set_odd_parity()`

Syntax

```
void ce_utils_des_set_odd_parity(DES_CBLOCK *key);
```

Parameters

key	Pointer to buffer containing DES key

Description

This function updates the DES key provided by the caller to ensure that it has odd parity.

Returns

None.

1.76 ce_utils_get_auth_params

API Name

`ce_utils_get_auth_params()`

Syntax

```
int ce_utils_get_auth_params(const char *name, int *len);
```

Parameters

name	Name of digest algorithm
len	Pointer to integer containing length of digest algorithm (output)

Description

This function returns the CryptoEngine numeric identifier (e.g., CE_ALG_MD5) associated with the specified digest algorithm (e.g., "MD5"). It also copies the digest length into '*len'.

Returns

This function returns the CryptoEngine numeric identifier (e.g., CE_ALG_MD5) associated with the specified digest algorithm (e.g., "MD5"). If the digest algorithm is not supported, it returns EFAILURE.

1.77 ce_x509cert_db_add

API Name

`ce_x509cert_db_add()`

Syntax

```
int ce_x509cert_db_add(CE_OP_PTR op, int domain, char *cert, char *prvkey,
char *passphrase)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
domain	Domain identifier
cert	Name of file containing X.509 certificate
prvkey	Name of file containing private key (corresponding to the public key stored in the certificate)
passphrase	Passphrase to decrypt (encrypted) private key file (NULL, if the private key file is in the clear)

Description

This function adds a X.509 certificate to the specified domain. It can also be used to update an existing entry.

Returns

This function returns ESUCCESS if the certificate was successfully added; otherwise, it returns EFAILURE.

1.78 ce_x509cert_db_create

API Name

`ce_x509cert_db_create()`

Syntax

```
int ce_x509cert_db_create(CE_OP_PTR op, int domain)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
domain	Domain identifier

Description

This function creates a new application-specific domain for the storage and processing of X.509 certificates. All certificates stored in a domain are considered trusted.

Returns

This function returns ESUCCESS if the domain was successfully created; otherwise, it returns EFAILURE.

1.79 ce_x509cert_db_del

API Name

`ce_x509cert_db_del()`

Syntax

```
int ce_x509cert_db_del(CE_OP_PTR op, int domain, char *cert)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
domain	Domain identifier
cert	Name of file containing X.509 certificate

Description

This function deletes a X.509 certificate from the specified domain.

Returns

This function returns ESUCCESS if the certificate was successfully deleted; otherwise, it returns EFAILURE.

1.80 ce_x509cert_db_destroy

API Name

`ce_x509cert_db_destroy()`

Syntax

```
int ce_x509cert_db_destroy(CE_OP_PTR op, int domain)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
domain	Domain identifier

Description

This function deletes the specified domain for the storage and processing of X.509 certificates.

Returns

This function returns ESUCCESS if the domain was successfully deleted; otherwise, it returns EFAILURE.

1.81 ce_x509cert_db_find

API Name

`ce_x509cert_db_find()`

Syntax

```
void *ce_x509cert_db_find(CE_OP_PTR op, int domain, char *cert)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
domain	Domain identifier
cert	Name of file containing X.509 certificate

Description

This function is used to locate a X.509 certificate in the specified domain.

Returns

This function returns a pointer to the matching certificate table entry. If a match cannot be located, it returns NULL.

1.82 ce_x509cert_db_list

API Name

`ce_x509cert_db_list()`

Syntax

```
int ce_x509cert_db_list(CE_OP_PTR op, int domain)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
domain	Domain identifier

Description

This function displays the Subject, Issuer, start date, and end date for each of the X.509 certificates in the specified domain.

Returns

This function returns ESUCCESS if the domain identifier is correct; otherwise, it returns EFAILURE.

1.83 ce_x509cert_free

API Name

`ce_x509cert_free()`

Syntax

```
void ce_x509cert_free(CE_OP_PTR op, void *cert)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
cert	Pointer to X.509 certificate structure

Description

This function frees the specified X.509 certificate data structure.

Returns

None.

1.84 ce_x509cert_get_pubkey

API Name

`ce_x509cert_get_pubkey()`

Syntax

```
void *ce_x509cert_get_pubkey(CE_OP_PTR op, void *cert);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
cert	Pointer to X.509 certificate structure

Description

This function returns a pointer to the public key structure associated with the Subject in the specified X.509 certificate.

Returns

Pointer to the public key associated with the Subject in the X.509 certificate.

1.85 ce_x509cert_get_subject_altname

API Name

`ce_x509cert_get_subject_altname()`

Syntax

```
int ce_x509cert_get_subject_altname(CE_OP_PTR op, void *cert, int pos, char
*altname, int *length, int *type)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
cert	Pointer to X.509 certificate structure
pos	Index of Subject Alternative Name (SAN) that is to be retrieved
altname	Pointer to output buffer for storage of retrieved SAN
length	Pointer to integer containing the length of the retrieved SAN (output)
type	Type of SAN that was retrieved (GEN_DNS, etc.)

Description

This function retrieves the n-th Subject Alternative Name specified in the X.509 certificate structure. (The desired index is specified via the 'pos' parameter.)

Returns

This function returns ESUCCESS if the requested parameter was successfully retrieved; otherwise, it returns EFAILURE.

1.86 ce_x509cert_getparm

API Name

`ce_x509cert_getparm()`

Syntax

```
int ce_x509cert_getparm(CE_OP_PTR op, void *cert, int param, u_char *buf,
int *len)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
cert	Pointer to X.509 certificate structure
param	Parameter being requested (Subject (CE_X509CERT_FIELD_SUBJECT or CE_X509CERT_FIELD_SUBJECT_DER), Issuer (CE_X509CERT_FIELD_ISSUER), start date (CE_X509CERT_FIELD_NOTBEFORE), end date (CE_X509CERT_FIELD_NOTAFTER))
buf	Output buffer
len	Pointer to integer containing the length of output buffer (input), and actual length required (output)

Description

This function writes the contents of the specified parameter in the X.509 certificate into the output buffer in text (or DER, for CE_X509CERT_FIELD_SUBJECT_DER) format. The textual data is not NULL-terminated.

Returns

This function returns ESUCCESS if the buffer was written to successfully; otherwise, it returns EFAILURE. The 'len' parameter is updated with the amount of data written (if the buffer was adequately long), or the required length of the buffer (if it wasn't).

1.87 ce_x509cert_print

API Name

`ce_x509cert_print()`

Syntax

```
int ce_x509cert_print(CE_OP_PTR op, void *cert, u_char *buf, int *len)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
cert	Pointer to X.509 certificate structure
buf	Output buffer
len	Pointer to the length of buffer (input), and actual length (output)

Description

This function writes the contents of the specified X.509 certificate into the output buffer in text format. The data is not NULL-terminated.

Returns

This function returns ESUCCESS if the buffer was written to successfully; otherwise, it returns EFAILURE. The 'len' parameter is updated with the amount of data written (if the buffer was adequately long), or the required length of the buffer (if it wasn't).

1.88 ce_x509cert_rd_buf

API Name

`ce_x509cert_rd_buf()`

Syntax

```
void *ce_x509cert_rd_buf(CE_OP_PTR op, u_char *buf, int len);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
buf	Buffer containing X.509 certificate in PEM format
len	Length of buffer

Description

This function reads a buffer containing a X.509 certificate, and creates the corresponding X.509 data structure.

Returns

This function returns a pointer to a X.509 data structure.

1.89 ce_x509cert_rd_buf2

API Name

`ce_x509cert_rd_buf2()`

Syntax

```
void *ce_x509cert_rd_buf2(CE_OP_PTR op, u_char *buf, int len);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
buf	Buffer containing X.509 certificate in DER format
len	Length of certificate (bytes)

Description

This function reads a buffer containing a X.509 certificate, and creates the corresponding X.509 structure.

Returns

This function returns a pointer to a X.509 structure.

1.90 ce_x509cert_rd_file

API Name

`ce_x509cert_rd_file()`

Syntax

```
void *ce_x509cert_rd_file(CE_OP_PTR op, char *name);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
name	File containing X.509 certificate in PEM format

Description

This function reads a file containing a X.509 certificate, and creates the corresponding X.509 data structure.

Returns

This function returns a pointer to a X.509 data structure.

1.91 ce_x509cert_rd_prvkey2

API Name

`ce_x509cert_rd_prvkey2()`

Syntax

```
int ce_x509cert_rd_prvkey2(CE_OP_PTR op, int domain, char *name, char
*passphrase, u_char *obuf, int *olen);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
domain	Domain identifier (e.g., CE_X509CERT_DOMAIN_IKE)
name	Name of file containing private key in PEM format
passphrase	Passphrase required to decrypt file (only if private key is encrypted)
obuf	Pointer to output buffer
olen	Pointer to integer containing length of DER-formatted private key (output)

Description

This function reads a file containing a private key, and copies the DER-formatted version into the output buffer provided by the caller. It also updates '*olen' with the amount of data written to 'obuf'.

Returns

This function returns ESUCCESS if the requested operation was performed successfully; otherwise, it returns EFAILURE.

1.92 ce_x509cert_store_add_trusted_cert

API Name

`ce_x509cert_store_add_trusted_cert()`

Syntax

```
int ce_x509cert_store_add_trusted_cert(CE_OP_PTR op, void *store, void
*cert)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
store	Pointer to certificate store
cert	Pointer to X.509 certificate

Description

This function adds a trusted certificate to the specified certificate store.

Returns

This function returns ESUCCESS if the certificate was successfully added; otherwise, it returns EFAILURE.

1.93 ce_x509cert_store_add_untrusted_cert

API Name

`ce_x509cert_store_add_untrusted_cert()`

Syntax

```
int ce_x509cert_store_add_untrusted_cert(CE_OP_PTR op, void *store, void
*cert)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
store	Pointer to certificate store
cert	Pointer to X.509 certificate

Description

This function adds an untrusted certificate to the specified certificate store. When adding multiple untrusted certificates (one at a time) via this function, the last certificate must belong to the entity whose identity is being verified.

Returns

This function returns ESUCCESS if the certificate was successfully added; otherwise, it returns EFAILURE.

1.94 ce_x509cert_store_create

API Name

`ce_x509cert_store_create()`

Syntax

```
void *ce_x509cert_store_create(CE_OP_PTR op, int domain)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
domain	Domain identifier

Description

This function creates a new certificate store for the validation of an entity's X.509 certificate. The validation process uses trusted certificates from the configured domain (specified at creation time) and those that are provided via ce_x509cert_store_add_trusted_cert().

Returns

This function returns a pointer to the newly created certificate store, or NULL if the store could not be created.

1.95 ce_x509cert_store_destroy

API Name

`ce_x509cert_store_destroy()`

Syntax

```
int ce_x509cert_store_destroy(CE_OP_PTR op, void *store)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
store	Pointer to certificate store

Description

This function deletes the specified certificate store.

Returns

This function returns ESUCCESS if the certificate store was successfully deleted; otherwise, it returns EFAILURE.

1.96 ce_x509cert_store_verify

API Name

`ce_x509cert_store_verify()`

Syntax

```
int ce_x509cert_store_verify(CE_OP_PTR op, void *store)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
store	Pointer to certificate store

Description

This function validates an (untrusted) entity's X.509 certificate by using the various trusted and untrusted certificates in the specified certificate store.

Returns

This function returns ESUCCESS if the entity's certificate is validated successfully; otherwise, it returns EFAILURE.

1.97 ce_x509cert_verify

API Name

`ce_x509cert_verify()`

Syntax

```
int ce_x509cert_verify(void *trusted_certs[], int tcount, void
*untrusted_certs[], int utcount)
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
trusted_certs	Array of pointers to trusted X.509 certificate data structures
tcount	Number of elements in the 'trusted_certs' array
untrusted_certs	Array of pointers to untrusted X.509 certificate data structures
utcount	Number of elements in the 'untrusted_certs' array

Description

This function verifies an entity's X.509 certificate using the various trusted and untrusted certificates. The last certificate in the set of untrusted certificates must belong to the entity whose identity is being verified.

Returns

This function returns ESUCCESS if the certificate was successfully deleted; otherwise, it returns EFAILURE.

1.98 ce_x509cert_wr_buf

API Name

`ce_x509cert_wr_buf()`

Syntax

```
void *ce_x509cert_wr_buf(CE_OP_PTR op, void *cert, u_char *buf, int len);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
cert	Pointer to X.509 certificate structure
buf	Buffer (to be written to)
len	Length of buffer

Description

This function writes the contents of the specified X.509 certificate data structure into the output buffer in DER format.

Returns

This function returns ESUCCESS if the buffer was written to successfully; otherwise, it returns EFAILURE.

1.99 ce_x509cert_wr_file

API Name

`ce_x509cert_wr_file()`

Syntax

```
void *ce_x509cert_wr_file(CE_OP_PTR op, void *cert, char *name, int format);
```

Parameters

op	Pointer to structure containing information about cryptographic operation requested (input) and status (output)
cert	Pointer to X.509 certificate structure
name	Name of file (to be written to)
format	Format of data written to file (DER (CE_FILETYPE_DER) or PEM (CE_FILETYPE_PEM))

Description

This function writes the contents of the X.509 data structure into the specified file in the requested (DER or PEM) format.

Returns

This function returns ESUCCESS if the file was written to successfully; otherwise, it returns EFAILURE.

2 Debugging

2.1 console_only

API Name

`console_only() - Stop all threads except console`

Syntax

```
void console_only(void *pio, bool_t dumpsystem)
```

Parameters

pio	Handle for output. If NULL, output goes to stdout.
dumpsystem	Non-zero means call the `dumpsysinfo` API before suspending tasks.

Description

Suspends all tasks, except the console. If dumpsystem is non-zero, it will call the `dumpsysinfo` API before suspending tasks. During debugging, an engineer could call this API when a special condition occurs, e.g., a dtrap. This API is only available when `NPDEBUG` is defined.

NOTE: The system cannot be returned to a normal state following this API.

Returns

- None

2.2 dprintf

API Name

`dprintf()`

`initmsg()`

Syntax

```
void dprintf(char *, ...);
```

```
void initmsg(char *, ...);
```

Description

These two routines are functionally the same as the standard C library `printf()` function and are called by the stack code to inform the porting engineer or end user of system status. `initmsg()` prints status messages at initialization time. `dprintf()` prints error warnings during runtime.

InterNiche provides an version of `printf()` in `misclib/ttyio.c`. This implementation does not support floating point formats, but is otherwise consistent with standard specifications for the function. The compile-time macro `NATIVE_PRINTF` (`ipport.h`) determines whether this code, or a user/library implementation is to be used.

See the detailed description in Debugging Aids.

2.3 dputchar

API Name

`dputchar() - Send a character to the console`

Syntax

```
void dputchar(int chr);
```

Description

The InterNiche CLI routines call `dputchar()` in order to display a character on the target system display or monitor port. If such output is not desired, `dputchar()` can be implemented as a no-op. Its parameter is an ASCII character that should be displayed on the target system display or monitor device.

`dputchar()` should perform newline expansion. If the value of `chr` is an ASCII newline character (`0xa`) then a newline followed by a carriage return should be output to the display device.

Returns

Nothing.

2.4 dtrap

API Name

`dtrap()`

Syntax

```
void dtrap(void);
```

Description

This primitive is intended to hook a debugger whenever it is called.

See the detailed description in the Debugging Aids section.

Returns

Usually nothing, depends on porting engineer modifications.

2.5 dumpsocketqs

API Name

`dumpsocketqs()`

Syntax

```
void dumpsocketqs(void *pio)
```

Parameters

pio	Handle for output. If NULL, output goes to stdout.

Description

Checks each allocated socket. If the socket's so_rcv.sb_cc or so_snd.sb_cc count is non-zero, then for each mbuf in the socket send or receive queue it will display the packet address, packet buffer length, mbuf address, mbuf data length, and for the send queue, the amount of data in the packet buffer that has already been acknowledged.

Returns

- None

2.6 dumpsysinfo

API Name

`dumpsysinfo()`

Syntax

```
void dumpsysinfo(void *pio)
```

Parameters

pio	Handle for output. If NULL, output goes to stdout.

Description

Calls several system state/status routines and prints the output to the stream specified by the pio parameter. It gives a fairly complete picture of memory and packet buffer usage, the state of all sockets in use, and the current status of all tasks including the driver.

Returns

- None

2.7 panic

API Name

`panic()`

Syntax

```
void panic(char *msg);
```

Parameters

```
char *msg /* short test message describing the fault */
```

Description

`panic()` is called if the InterNiche stack software detects a fatal system error. `msg` is a string describing problem. What this should do varies with the implementation. In a testing or development environment it should print messages, hook debuggers, etc. In an embedded controller, it should try to restart (i.e. warm boot) the system.

Sample for a DOS application is shown below.

Returns

Generally there is no return from this routine, however it is sometimes useful to allow a return under control of a debugger.

Example

```
void
panic (msg)
   char *msg;
{
   dprintf("panic: %s\n", msg);
   dtrap();      /* try to hook debugger */
   netexit(1);   /* try to clean up */
}
```

3 Device

3.1 SignalPktDemux

API Name

`SignalPktDemux()`

Syntax

```
void SignalPktDemux(void);
```

Description

`SignalPktDemux()` is called by network interface code to indicate to the InterNiche IP layer that received packets have been enqueued to `rcvdq`. A call to `SignalPktDemux()` should result in the target system calling the portable `pktdemux()` function to dequeue `rcvdq`.

The implementation of `SignalPktDemux()` is dependent upon whether the target system has a multitasking OS or is implemented as a superloop. If the target system is implemented as a superloop, as is the case in the `w32_in_vc` reference port, `SignalPktDemux()` can be a no-op so long as the superloop calls `pktdemux()` on a regular basis. No explicit notification is necessary in this case.

If the target system has a multitasking OS, `SignalPktDemux()` could still be a no-op so long as a task was created that periodically called `pktdemux()` to empty `rcvdq`. This approach can be made to work but is sub-optimal in systems with a multitasking OS. The preferred approach in this case is to create a task that is constructed as a loop in which, on each pass through the loop, the task blocked on some OS dependent event. Upon return from the event block, the task would call `pktdemux()`. In this case, `SignalPktDemux()` would post the event on which the task was blocked so as to cause the task to call `pktdemux()`. This is shown below:

Example

```
void receiveTask()
{
   for ( ; ; )
   {
      block on event X;   /* this call will be dependent on the OS */
      pktdemux();         /* portable function to empty rcvdq */
   }
}

void SignalPktDemux()
{
   post event X;          /* this call will be dependent on the OS */
}
```

3.2 clock_init

API Name

`clock_init - Start the NicheStack clock`

Syntax

```
void clock_init(void);
```

Parameters

None.

Description

This sets up a periodic call to the function in_tick_hook() at a rate of 100Hz (defined as PPS). It is possible that the OS timer is faster than this, and there is a provision for skipping n calls before ticking.

If the RTOS does not provide a time base then the port must provide one and handle the irq routing to get in_tick_hook called.

If the NicheStack clock is disabled, reset all NicheStack clock variables to their initial values and enable the NicheStack clock. If the NicheStack is already enabled, calls to `clock_init( )` do nothing. A call to `clock_c( )` to disable the NicheStack clock is required before the clock can be reenabled.

Returns

Nothing

3.3 emac_close

API Name

`emac_close() - Shutdown Ethernet device`

Syntax

```
int emac_close(int index)
```

Parameters

`index`	interface number in the nets array

Description

Disable the ethernet hardware, and free any allocated resources

Returns

Success(0) or failure (1)

3.4 emac_core_enable

API Name

`emac_core_enable - Enable HW block for ethernet`

Syntax

```
int emac_core_enable(IN_ETH eth)
```

Parameters

`eth`	Ethernet control block for the intended inteface

Description

Enable the hardware block for the ethernet module, setup any pointers to the block that may need to be remembered in the IN_ETH structure.

Returns

Success(0) or failure (1)

3.5 emac_hw_init

API Name

`emac_hw_init - Start ethernet interface traffic`

Syntax

```
int emac_hw_init(IN_ETH eth)
```

Parameters

`eth`	Ethernet control block for the intended inteface

Description

Enable the hardware to start transfering packets, inluding enabling the interrupts. This function is called from the common ethernet initialization code.

Returns

Success(0) or failure (1)

3.6 emac_mac_init

API Name

`emac_mac_init() - Ethernet interface config`

Syntax

```
int emac_mac_init(IN_ETH eth)
```

Parameters

`eth`	Ethernet control block for the intended inteface

Description

Initialize the hardware block for operation. This function is called from the common ethernet initialization code.

Returns

Success(0) or failure (1)

3.7 emac_phy_read

API Name

`emac_phy_read() - Read PHY register`

Syntax

```
unsigned short emac_phy_read(IN_ETH eth, unsigned phyaddr, unsigned phyreg)
```

Parameters

`eth`	Ethernet control block for the intended inteface
`phyaddr`	address of the PHY
`phyreg`	address of the register to read

Description

Read a register from a ethernet PHY attached to this ethernet block

Returns

16 bit word read from PHY

3.8 emac_phy_write

API Name

`emac_phy_write() - Write PHY register`

Syntax

```
void emac_phy_write(IN_ETH eth, unsigned phyaddr, unsigned phyreg, const
unsigned short data);
```

Parameters

`eth`	Ethernet control block for the intended inteface what it does
`phyaddrvar1`	address of the phy what it does
`phyregvar2`	address of the register to write what it does
`data`	Value to write to the PHY

Description

Write a register from a ethernet PHY attached to this ethernet block.

Notes/Status

Returns

Nothing

3.9 emac_rxtx_init

API Name

`emac_rxtx_init()- Setup ethernet buffer usage`

Syntax

```
int emac_rxtx_init(IN_ETH eth)
```

Parameters

`eth`	Ethernet control block for the intended inteface

Description

Initialize the rx and tx descriptors and packet queues. This function is called from the common ethernet initialization code.

Returns

Success(0) or failure (1)

3.10 emac_send

API Name

`emac_send() - Process the outgoing packet queue`

Syntax

```
void emac_send(IN_ETH eth);
```

Parameters

`eth`	Ethernet control block for the intended inteface

Description

Invoke the send logic to see if any of the queued packets are able to be passed to the hardware. This routine is called from the common ethernet code from within the function called by `net->pkt_send`.

Notes/Status

Returns

Nothing

3.11 eth_setlink

API Name

`eth_setlink()- Set ethernet link status`

Syntax

```
void eth_setlink(IN_ETH eth, unsigned phyaddr, int speed, bool_t duplex)
```

Parameters

`eth`	Ethernet control block for the intended inteface
`phyaddr`	address of the phy
`speed`	speed of connection
`duplex`	Full duplex(true) or half (false)

Description

Inform the network driver about a change in link status. This is called from the phy link change ISR and from the init routines.

Returns

Nothing

3.12 get_cticks

API Name

`get_cticks() - Get slow timer tick count`

Syntax

```
uint32_t get_cticks(void)
```

Parameters

None.

Description

Counts time since power up. Frequency is TPS (default 20Hz)

Returns

Returns 32-bit tick count of time since power up.

3.13 getch

API Name

`getch() - Get character from console`

Syntax

```
int getch(void);
```

Description

`kbhit()` and `getch()` are used together to effect CLI input. The stack code calls `kbhit()` to determine if a character is available and then if a character is available, calls `getch()` to return the value of the character. **`getch()` should never block for user input**.

Returns

If a character is available at the CLI or system monitor device, `getch()` returns the ASCII value of that character. Its return value is undefined if no character is available.

3.14 kbhit

API Name

`kbhit() - Pool for character ready from console`

Syntax

```
int kbhit(void);
```

Description

`kbhit()` should return a non-zero value if a keystroke has been entered by a user at the CLI of the target system. It should not dequeue the character itself from the input device, rather the return value from `kbhit()` should simply poll the device to determine if a character is present. The entered character is retrieved using the `getch()` function.

Returns

`0` if no character had been entered at the input monitor device, non-zero if at least one character is available.

3.15 n_close

API Name

`n_close() - Accessor to shutdown network interface`

Syntax

```
int n_close(int if_number);
```

Parameters

```
int if_number /* index into nets[ ] for NET to close */
```

Description

Does whatever is necessary to restore the device and its associated driver software prior to exiting the application. This function may not be required to do anything on embedded systems which start their devices at power up and don't have any reason to shut them down. If packet types (i.e.: `0x0800` for IP and `0x0806` for ARP) have been accessed in a lower layer driver, they should be released here.

Returns

Returns `0` if OK, else one of the `ENP_` codes.

3.16 n_init

API Name

`n_init() - Accessor to startup network interface`

Syntax

```
int n_init(int if_number);
```

Parameters

```
int if_number /* interface number, for indexing nets[ ] */
```

Description

This routine is responsible for preparing the device to send and receive packets. It is called during system startup time after `prep_ifaces()` has been called, but before any of the other network interface's routines are invoked. When this routine returns, the device should be set up as follows:

- Net hardware ready to send and receive packets.
- All required fields of the net structure are filled in.
- Interface's MIB-II structure filled in as show below.
- IP addressing information should be set before this returns unless DHCP or BOOTP is to be used. See the section titled **"Initialization of net Structure IP Addressing Fields**".

This will usually include hardware operations such as initializing the device and enabling interrupts. It does not include setting protocol types. This is handled later (see the section **n_reg_type**. Upon returning from this routine it is safe for your hardware's interrupt or receive routines to start enqueuing received packets in the `rcvdq`. Packets which are not IP or ARP will be discarded by the stack.

The `nets[ ]` structure array element that is indexed by `if_number` should be completely filled in when this function returns. Note that the work of filling this structure is shared between `prep_ifaces()` and this function, so if all `nets[ ]` structure setup was done in `prep_ifaces()` (see **The 'glue' Layer**) there may be nothing to do here.

Shown below is an example of code that can be used for setting up the MIB structure for a 10 Mbps Ethernet interface. The n_mib field of the nets[ ] structure points to a structure that is used to contain the MIB information which has already been statically allocated by the calling code. See RFC1213 for detailed descriptions of the MIB fields. Most of the MIB fields are used only for debugging and statistical information, and are not critical unless your device is managed by SNMP. The ifPhysAddress field is an exception. It is used by ARP to obtain the hardware's MAC address and MUST be set up correctly for the IP stack to work over Ethernet. Note that although ifPhysAddress is a pointer, it does not point to valid memory when the MIB structure is created. The porting engineer should make sure it points to a static buffer containing the MAC address before this function returns. The size of this address is determined by the media (6 bytes for Ethernet) and should be set in the nets[ ] structure member n_hal (hardware address length).

```
u_char macaddress[6];   /* should contain interface's MAC address */

nets[if_number]->n_mib->ifDescr = "Ethernet Packet Driver";
nets[if_number]->n_mib->ifType = ETHERNET; /* SNMP Ethernet type */
nets[if_number]->n_mib->ifMtu = ET_MAXLEN;
nets[if_number]->n_mib->ifSpeed = 10000000; /* 10 megabits per second */
nets[if_number]->n_mib->ifAdminStatus = 1;
nets[if_number]->n_mib->ifOperStatus = 1;
nets[if_number]->n_mib->ifPhysAddress = ...macaddress[0]; /* example */
```

Returns

Returns 0 if OK, else one of the ENP_ codes.

3.17 n_refill

API Name

`n_refill - Replenish device driver's internal resources.`

Syntax

```
void (*n_refill)(int iface);
```

Parameters

`iface`	interface number of device to refill

Description

Refills the device's internal packet buffer pool by calling `PK_ALLOC()` or `PK_CONTIG` to obtain packet buffers from the Stack's free packet buffer queues. The 'iface' parameter specifies the index of the device in the '`nets[iface]`' array. The `FREEQ_RESID` resource should be locked within the '`n_refill`' function prior to allocating the packets. The number of packets and their sizes is dependent upon the design of the driver. If there are multiple devices in the system, the developer can implement a single '`n_refill`' function for all of the devices or a separate '`n_refill`' function for each device.

The '`n_refill`' function is called in the `pktdemux()` function which is normally part of the main NicheStack task. The 'n_refill' function should no consider it an error if the device's internal packet buffer pool cannot be completely refilled.

Returns

Nothing

3.18 n_reg_type

API Name

`n_reg_type()`

Syntax

```
int n_reg_type(unshort type, NET net);
```

Parameters

`unshort type NET net`

Description

Register with any lower level drivers to receive a MAC type, i.e. `0x0800` for IP and `0x0806` for ARP. On most embedded systems with Ethernet, where the InterNiche stack does not share the hardware with other network stacks, no action is required. Since the InterNiche stack gets all the packets anyway, `n_reg_type()` can simple return an `OK` code without doing anything. The porting engineer should be sure, however, that all received packets will be passed to the stack. Note that on some driver subsystems a type must be registered with the driver informing it that we are interested in the packets.

On SLIP links, all packets are IP, so nothing has to be done in `n_reg_type()`.

On PPP links, PPP will sort out the packets, so again, nothing has to done in `n_reg_type()`.

Returns

Returns `0` if OK, else one of the `ENP_` codes.

3.19 n_stats

API Name

`n_stats() - Accessor to get network interface statistics`

Syntax

```
int (*n_stats)(int iface, void *stats);
```

Parameters

```
int iface /* interface number to dump statistics for */

void * stats /* pointer to a user defined structure */
```

Description

OPTIONAL: `n_stats()` enables the driver to provide hardware specific information which is not included in the generic MIB-II interface group. This information might include hardware specific error counters, such as the number of collisions on an Ethernet link; or internal resource information, such as the status and number of current buffers available on a ring-buffer device. The definition of the '`stats`' structure and its contents is left to the driver writer. An example is the `enet_stats` structure in h `/ether.h`.

Returns

The function returns `ESUCCESS` if it is successful and `EFAILURE` if an error, such as a parameter value out of range, is encountered.

4 ESMTP

4.1 esmtp_attachother

API Name

esmtp_attachother - Create a Non-ASCII Attachment

Syntax

```
int esmtp_attachotherbuf(int ssid, uint8_t *buf, char *displayname, int
numbytes,char *mimetype, char *mimesubtype, char *mimeparam);

int esmtp_attachotherfile(int ssid, char *filename, char *displayname, char
*mimetype, char *mimesubtype, char *mimeparam);

int esmtp_attachotherfunc(int ssid, ATTACHDATACB funcptr, char
*displayname, char *mimetype, char *mimesubtype, char *mimeparam);
```

Parameters

ssid	value returned by `esmtp_startsession()`
buf	Pointer to an application provided buffer. The buffer and the data must remain constant until the email has been accepted by the email server as indicated by a call the application's main callback function.
filename	Pointer to the name of a file that ESMTP can open and read. In the NicheStack demo for Windows, the file must be in the same directory as iniche.exe (`src41/ReferencePorts/w32_nichetask_vs`).
funcptr	Pointer to an application provided callback function. While the email session between ESMTP and the email server is in progress, ESMTP will call this one or more times to obtain the data for the body or an attachment (See description of `BODYDATACB()` and `ATTACHDATACB()`)
displayname	Suggested display name. It should not contain path information. If the attachment data comes from a file, the display name need not be the same as the source file name. The sending application can only suggest a name to be used. It cannot suggest path information. The email display program on the final delivery system makes the determination of the name displayed for an attachment.
mimetype	Pointer to a NUL terminated string name for one of the top-level Media Types as described in RFC 2046. Other types are possible as long they are understood by the specified email server. Examples: "`text`", "`image`", "`Application`"
mimesubtype	Pointer to a NUL terminated string name for a Media subtype. Many subtypes are possible as long they are understood by the specified email server. Examples: `gif`, `octet-stream`.
mimeparm	NULL or a pointer to a NUL terminated string that adds a MIME-type qualifier to the mimetype and subtype. For example, it may specify a language or a character set.

Description

These APIs tell ESMTP how to obtain the data for one email attachment. An email may have multiple attachments, so any of these APIs may be called multiple times.

Examples:

```
esmtp_attachotherfile(3, "pic123", "yourpicture.jpg", image, jpeg, NULL);
esmtp_attachotherfunc(4, sensorfunc, "currdata", "application", "octet-stream",
```

Returns

0 for success or one of the negative ESMTP error codes.

4.2 esmtp_attachtext

API Name

`esmtp_attachtext - Create an ASCII Attachment`

Syntax

```
int esmtp_attachbuftext(int ssid, char *buf, char *displayname);

int esmtp_attachfiletext(int ssid, char *filename, char *displayname);

int esmtp_attachfunctext(int ssid, ATTACHDATACB funcptr, char
*displayname);
```

Parameters

ssid	Value returned by `esmtp_startsession()`.
buf	Pointer to an application provided buffer containing ASCII data. The end of the data is indicated by NUL. There cannot be any other NULs or non-ASCII data before this NUL. The buffer and data must remain constant until the email has been accepted by the email server as indicated by a call to the application's main callback function.
filename	Pointer to the name of a file that ESMTP can open and read. In the NicheStack demo for Windows, the file must be in the same directory as `iniche.exe` (example: "`src41/ReferencePorts/w32_nichetask_vs`").
funcptr	Pointer to an application provided callback. While the email session between ESMTP and the email server is in progress, ESMTP calls this once or more to obtain the data for the body or an attachment (See BODYDATACB and ATTACHDATACB).
displayname	Suggested display name. It should not contain path information. If the attachment data comes from a file, the display name need not be the same as the source file name. The sending application can only suggest a name to be used. It cannot suggest path information. The email display program on the final delivery system makes the final determination of the name displayed for an attachment.

Description

These APIs tell ESMTP how to obtain the data for 1 email attachment. There is a separate API for each of the three possible sources of data. The second parameter is a pointer to a specific source type. An email may have multiple attachments, so any of these APIs may be called multiple times.

Returns

0 for success or one of the negative ESMTP error codes.

4.3 esmtp_body

API Name

```
esmtp_body - Create the message's "body"
```

Syntax

Data from buffer

```
int esmtp_bodybuftext(int ssid, char *bufptr);
```

Data from file

```
int esmtp_bodyfiletext(int ssid, char *filename);
```

Data from Function

```
int esmtp_bodyfunctext(int ssid, BODYDATACB funcptr);
```

Data from Command

```
int esmtp_bodyclicmdtext(int ssid, char *cmdstr);
```

Data from script

```
int esmtp_bodyscripttext(int ssid, char *scriptname);
```

Parameters

ssid	Value returned by `esmtp_startsession()`.
bufptr	Pointer to an application provided buffer containing ASCII data. The end of the data is indicated by a NUL. There cannot be any other NULs or non-ASCII data preceding this NUL. The buffer and the data must remain constant until the email has been accepted by the email server as indicated by a call to the application's main callback function.
filename	Pointer to the name of a file that ESMTP can open and read. In the NicheStack demo for Windows, the file must be in the same directory as `iniche.exe` (e.g.: "`src41 /ReferencePorts/w32_nichetask_vs`").
funcptr	Pointer to an application provided callback function. While the email session between ESMTP and the email server is in progress, ESMTP will call this one or more times to obtain the data for the body (See description of `BODYDATACB()`)
cmdstr	Pointer to a NUL terminated CLI command string where the string is a command name plus any parameters exactly as it would be typed at a console. This string must remain constant until the email has been accepted by the email server as indicated by a call the application's main callback function. **This is only available for the message body**
scriptname	Pointer to the name of a script file that ESMTP can open and read. This file contains one or more CLI command strings. **This is only available for the message body**

Description

These APIs tell ESMTP how to obtain the data for the body of an email. There is a separate API for each of the five possible sources of data. The second parameter of each API is a pointer a specific type of source. A single email can only have one body, so only one of these calls can be used with each email. In all cases, the body can only contain ASCII data.

Returns

0 for success or one of the negative ESMTP error codes.

4.4 ATTACHDATACB

API Name

`ATTACHDATACB() - Callback function that produces data for an attachment`

Syntax

```
int (*ATTACHDATACB)(int ssid, char *buf, int *len);
```

Parameters

ssid	SSID for this session
buf	ESMTP provided buffer where the callback function should write the data
len	At invocation, the maximum size of 'buf'. Upon return, the number of bytes put in 'buf' by application

Description

This application callback function is passed by the esmtp_attachfunctext() API if it will produce only ASCII data, or by the esmtp_attachotherfunc() API if it will produce non-ASCII data.. During email session, after the email body has been sent to the email server, ESMTP will call this application function to obtain the data for an email attachment. The function should write from 0 to “len” bytes of data into the provided buffer and set the “len” parameter to the number of bytes written.

Notes

- The callback **must** update (`* len`) in addition to providing the proper return value.

Returns

- `ESM_CALLAGAIN`: After it has sent the data from this call, ESMTP should call the function again to obtain more data. ESMTP will call the function again at the next opportunity, even if no data was provided in this call.
- `0`: All data has been passed. Do not call again.

4.5 BODYDATACB

API Name

`BODYDATACB() - Callback function that produces ASCII data for the message body`

Syntax

```
int (*BODYDATACB)(int ssid, char *buf, int *len);
```

Parameters

ssid	SSID for this session
buf	ESMTP provided buffer where the callback function should write the data
len	At invocation, the maximum size of 'buf'. Upon return, the number of bytes put in 'buf' by application

Description

This application callback function is passed to ESMTP by the esmtp_bodyfunctext() API. After an email session has been opened to the email server and the email headers have been sent, ESMTP will call this application function to obtain the data for the email body. The function should write from 0 to “len” bytes of ASCII data into the provided buffer and set the `len` parameter to the number of bytes written.

Notes

- The callback **must** update (`* len`) in addition to providing the proper return value.

Returns

- `ESM_CALLAGAIN`: After it has sent the data from this call, ESMTP should call the function again to obtain more data. ESMTP will call the function again at the next opportunity, even if no data was provided in this call.
- `0`: All data has been passed. Do not call again.

4.6 cb_func

API Name

`cb_func() - Required application callback function used by ESMTP to report the final status on an email session`

Syntax

```
void (*cb_func)(int, int, int, void *));
```

Parameters

int ssid	SSID for this session
int status	Status for this email session. Defined value are (see esmtp_port.h):

define name	value	description
ESM_TYPE_FATAL	1	An error occurred which will cause the session to close. The email was not delivered
ESM_TYPE_MAILDONE	2	Email session closed after the email server accepted the email. Server responsible for delivery
ESM_TYPE_CLOSED	3	Email session closed without the email server accepting the email. The email will not be delivered.

int error	Zero for success or one of the ESMERR codes defined in esmtp_port.h
void *data	Usually NULL, but may contain a pointer to a string describing one or more non-fatal errors.

Description

This callback is a required parameter for the esmtp_start API. It is used to report the results of the email session: error value, email delivered, or session closed without the email being delivered. Most ESMTP errors are fatal and will cause the email session to close. However, errors related to individual email recipients (improperly formatted, rejected by email server, etc.) are not fatal as long as there is at least one valid recipient. When called, the cb_func data parameter may contain a pointer to a string that gives information about one or more errors related to recipients.

Returns: Nothing

4.7 esmtp_exec

API Name

`esmtp_exec() - Execute the email command`

Syntax

```
int esmtp_exec(int ssid);
```

Parameters

`ssid`	value returned by `esmtp_startsession()`

Description

This API tells ESMTP that the user has finished passing all parameters for this email session and ESMTP should send the email. ESMTP will:

1. open a session with the specified email server
2. use the SMTP protocol to pass all of the header and data information to the email server. Obtaining the data for the email body may require opening and reading a specified file or making one or more calls to a callback function that produces email body data.
3. Call the session cb_func to report the final status of the email.

Returns

0 for success or one of the negative esmtp error codes.

4.8 esmtp_param

API Name

`esmtp_param() - Store a single parameter (from, to, etc) for current mail session.`

Syntax

```
int esmtp_param(int ssid, uint16_t type, char *param);
```

Parameters

ssid	value returned by esmtp_startsession()
type	One of the first 6 non-data parameter types defined in `esmtp_port.h`: • ESMTP_FROM • ESMTP_REPLYTO • ESMTP_TO • ESMTP_CC • ESMTP_BCC • ESMTP_SUBJECT
param	String containing a single parameter.

Description

This API is used to pass parameters used in the header fields of the email message. Only one parameter may be passed with each call. All addresses must be in the form: mailbox@domain name (e. g. emailname@yahoo.com). A separate call must be made to pass each address. All of the calls that take an address may be called multiple times.

The call is repeated for each parameter.

Notes:

1. The address used for ESMTP_FROM must be a registered user on the email server specified in the esmtp_startsession() call.
2. The must be at least one call to ESMTP_FROM and there must be at least one recipient.

Returns

0 for success or one of the negative esmtp error codes.

4.9 esmtp_quitbyssid

API Name

`esmtp_quitbyssid() - Close (abort) an active email session`

Syntax

```
int esmtp_quitbyssid(int ssid);
```

Parameters

`ssid`	value returned by esmtp_startsession()

Description

This API is used to abort an active email session. It may be used at any time between the call to esmtp_startsession() and ESMTP's call to the cb_func that indicates the final status of the email session. If ESMTP has already opened an SNTP connection to the email server, it will send an SMTP QUIT command. All session memory will be freed as a result of this command.

Calling this API after ESMTP's call to the application's callback function will result in a ESMTP_CONN_NOTFOUND error.

Returns

0 for success or one of the negative esmtp error codes.

4.10 esmtp_startsession

API Name

`esmtp_startsession() - Start an ESMTP session to send one email`

Syntax

```
int esmtp_startsession(char *server, char *port, uint32_t flags, char
*username, char *password, void (*cb_func)(int, int, int, void *));
```

Parameters

server	domain name of email server
port	port to use on email server
flags	bit field where the bits represent features for this email session

ESMCF_USESSL	0x01	Use SSL to connect to server
ESMCF_USEAUTH	0x02	Authenticate via usernames and passwords
ESMCF_IP4	0x04e	Prefer IPv4 for this connection to server
ESMCF_IP6	0x08	Prefer IPv6 for this connection to server

username	user name for users account on specified email server
password	password for users account on specified email server
cb_func	application callback function that ESMTP module will use to report final email status

Description

This routine is used to pass the basic parameters for an email session. ESMTP will:

- validate the parameters
- alloc the required ESMPT memory pools if they do not already exist
- Use DHCP to obtain an IP address for the specified server
- Save the session parameters in the session memory pool
- Return the SSID to be used with all APIs and callback functions.

Returns

Positive Session ID (SSID) or one of the negative error codes listed in esmtp_port.h

5 FTP Client

There is just one function:

Command Name

`FTP client API`

Description

The FTP Client can be entirely controlled by a user-developed application program. The following tables show the relation between FTP Client commands and FTP Client APIs. The file `ftpctest.c` provides basic examples for calling and checking the results of each of these APIs.

Syntax

`ascii`	Use ASCII transfer mode
`int fc_settype(struct ftpc *ftpconn, int type);`	

`binary`	Use binary transfer mode
`int fc_settype(struct ftpc *ftpconn, int type);`	

`cd`	Change directory on the server
`int fc_chdir(struct ftpc *ftpconn, char *dirparm);`	

`delfile`	Delete a file on the server
`int fc_delfile(struct ftpc *ftpconn, char *dirparm);`	

`fquit`	Close the FTP session and connection
`int fc_quit(struct ftpc *ftpconn);`	

`ftp`	Allocates and initializes a struct ftpc. Opens an FTP Client connection to an FTP server. Returns a pointer to the struct ftpc for the connection.
`ftpc *fc_connect(void *fhost, char *user, char *passwd, void *pio, int domain);`	

get	GET a file (transfer file from server directory)
int fc_get(struct ftpc *ftpconn, char *fname, char *lname);	

ls	List files in the server directory
int fc_dir(struct ftpc *ftpconn);	

mkdir	Create a new directory on the server
int fc_mkdir(struct ftpc *ftpconn, char *dirparm);	

pasv	Set FTP server to passive mode
int fc_pasv(struct ftpc *ftpconn);	

put	Put a file (transfer file to server directory)
int fc_put(struct ftpc *ftpconn, char *fname, char *lname);]	

pwd	Print the current working directory on the server
int fc_pwd(struct ftpc *ftpconn);	

rnamefile	Rename an existing file on the server
int fc_rnamefile(struct ftpc *ftpconn, char *fromname, char *toname);	

rmdir	Remove (delete) a directory on the server
int fc_put(struct ftpc *ftpconn, char *fname, char *lname);]	

Parameters

`struct ftpc *`	Pointer to the struct ftpc for the connection (see `ftpclnt.h`). The structure contains all of the information that the FTP Client retains concerning the connection. A pointer to this structure is returned by the `fc_connect()` function, and it is the first parameter for all other FTP Client APIs.
`int type`	`FTPTYPE_ASCII` ("ascii") or `FTPTYPE_IMAGE` ("binary"). See `ftpsrv.h`.
`char * dirparm`	Pointer to character string containing a directory name on the FTP server.
`char * fname`	Pointer to character string containing a file name (an optionally a path) on the FTP server.
`char * lname`	Pointer to character string contain a file name (an optionally a path) on the FTP Client system.
`fhost`	pointer to binary IP address of the FTP server in big endian format
`user`	Pointer to string containing the user name for the connection
`passwd`	Pointer to string containing the password for the connection
`pio`	pointer to a Generic I/O Structure. If the pointer is NULL, then any output is sent to standard out domain AF_INET for IPv4 AF_INET6 for IPv6

Notes/Status

- FTP client commands which correspond to the API exist as individual menus, rather than members of an overall FTP client menu. For this reason, it is often necessary to type "FTP" in front of the command name in order to disambiguate it from another command, e.g., you must type "FTP put", rather than simply "put".
- All pointers to a file name or a directory name may include a path. This path can be absolute or relative to the current working directory. For fc_get() and fc_put(), the current working directory is local for lname and remote for fname.
- If FC_USECALLBACK is defined in ftpclnt.h, then the FTP Client task will make a call to ftpc_callback() each time there is a change in the FTP Client state. The FTP Client states are listed and described in ftpclnt.h. There are 6 client login states that culminate in the FCL_LOGGEDIN state. This can be considered as the idle state for the FTP client. Once the user has logged into the FTP server, then every subsequent command/API starts and finishes in the FCL_LOGGEDIN state (except for fc_quit which ends with the freeing of the struct ftpc for the connection).
- The example test program in ftpctest.c contains a function ftpc_waitstate(). This example function is called following each FTP Client API call. It polls the fc->logstate waiting for the state to return to FCL_LOGGEDIN, which signifies that the command has completed. At that point it tests to see if the last response code received from the FTP server (fc->last_rcode) matches the value expected for the command. Again, this is only an example. The porting engineer should write a function that meets the specific needs of the controlling application.
- It should be noted that with your application, more than one response code may be acceptable for a specific API. An example is the fc_mkdir() API. A response code of 550, which indicates that the directory already exists on the server, may, or may not be acceptable for your application.
- The file ftpcport.c contains a default implementation of the ftpc_callback() function, designed to support the FTP Client test program running on a system that contains a console. When the FTP Client is controlled by a user-developed application, the porting engineer should modify the callback function to meet the needs of the controlling application.

6 GIO

6.1 gio_dev

API Name

`gio_dev() - initialize a GIO context`

Syntax

```
GIO *gio_dev(GIO *gio, void *id, GIO_FUNC in, GIO_FUNC out, int rw, int
type)
```

Parameters

gio	pointer to the GIO context
id	device id
in	input function
out	output function
rw	update flags: GIO_R = update input stream fields GIO_W = update output stream fields GIO_RW = update input and output stream fields
type	device type code

Description

If `GIO_R` is set in the '`rw`' field, update the input stream with the '`id`', '`in`', and '`type`' values. If `GIO_W` is set in the '`rw`' field, update the output stream with the '`id`', '`out`', and '`type`' values.

Prototypes for generic device-specific input and output functions can be found in gio.h.

Returns

A pointer to the updated `GIO` context is returned.

6.2 gio_done

API Name

`gio_done() - call the GIO callback function`

Syntax

```
int gio_done(GIO *gio, int32_t code)
```

Parameters

gio	pointer to the GIO context
code	"done" code

Description

Call the GIO context's 'done' function. The syntax of the callback is:

```
ret = (gio->done)(gio, gio->param, code);
```

Returns

The return value from the '`done`' function is an integer completion code, where 0 means success and non-zero is a user-defined error code.

6.3 gio_in

API Name

`gio_in() - read data from the GIO input stream`

Syntax

```
int gio_in(GIO *gio, char *buf, uint32_t len)
```

Parameters

gio	pointer to the GIO context
buf	input buffer pointer
len	maximum input length

Description

Read a maxmimum of 'len' bytes of data from the GIO input stream. The data is stored in the buffer pointed to by 'buf'. If the `GIO_F_BIN` flag is set in the GIO context, the function blocks until 'len' bytes are read. If `GIO_F_BIN` is not set, the function returns immediately after copying any available data into the buffer.

Returns

If the return value is positive, the return value is the number of bytes read from the device. A return value of zero means there is no data available. A negative return value indicates an error occurred.

6.4 gio_out

API Name

`gio_out() - write data to the GIO output stream`

Syntax

```
int gio_out(GIO *gio, char *buf, uint32_t len)
```

Parameters

gio	pointer to the GIO context
buf	output buffer pointer
len	output buffer length

Description

Write 'len' bytes of data to the GIO output stream. The 'buf' parameter points to the first byte of data to be written. If the GIO_F_BOUT flag is set in the GIO context, the function blocks until 'len' bytes are written. If GIO_F_BOUT is not set, the function returns immediately after copying up to 'len' bytes into the buffer.

Returns

If the return value is positive, the return value is the number of bytes written to the device (possibly zero bytes). A negative return value indicates an error occurred.

If fewer than 'len' bytes were written to the output stream, the caller should update the buffer pointer and remaining byte count, and wait and retry the operation.

6.5 gio_pop

API Name

`gio_pop() - "pop" the current GIO context`

Syntax

```
int gio_pop(GIO **giop);
```

Parameters

giop	address of a pointer to the GIO context

Description

The current GIO context (pointed to by the GIO context variable pointed to by 'giop') is destroyed, and the GIO context pointer pointed to by the 'giop' parameter is updated to point to the previous GIO context.

Returns

The return code is a GIO error code defined in `gio.h` indicating the success or failure of the operation.

6.6 gio_printf

API Name

`gio_printf() - write formatted data to the GIO output stream`

Syntax

```
int gio_printf(GIO *gio, const char *format, ...);
```

Parameters

gio	pointer to the GIO context
format	printf()-compatible format specification string
...	output parameter list

Description

Creates a formatted output string using the format specification and the output parameter list. The formatted output string is then written to the GIO output stream. This function is equivalent to:

```
char buf[N];

sprintf(buf, format, ...);
ret = gio_out(gio, buf, strlen(buf));
return (ret);
```

The gio_out() operation is forced to be performed in blocking I/O mode (GIO_F_BOUT is set).

Returns

Function returns the result of the `gio_out()` call (see above).

6.7 gio_push

API Name

`gio_push() - "push" the current GIO context`

Syntax

```
int gio_push(GIO **giop, void *id, GIO_FUNC in, GIO_FUNC out, int rw, int
type)
```

Parameters

giop	pointer to the GIO context
id	device id
in	input function
out	output function
rw	update flags: GIO_R = update input stream fields GIO_W = update output stream fields GIO_RW = update input and output stream fields
type	device type code

Description

Similar to the `gio_dev()` function except that the current GIO context, pointed to by *giop, is saved and a new GIO context is created. The new GIO context is initialized with the values of the current GIO context, and then gio_dev() is called to update the new context with the function parameters.

`gio_push()` is used to change an existing context. If you were to use `gio_dev()` to change an existing context, then the previous context would be lost. For example, with `gio_push()`, if you change from reading from a socket to reading from a file, when you perform a `gio_pop()`, the application will again read input from the original socket.

Note: The "id" parameter is simply a pointer that is passed to the input and output routines. For example:

```
err = gio_push(&hp->ctx->gio,
               (void *)&hp->si,
               &wbs_io_in,
               &wbs_io_out,
               GIO_RW,
               GIO_SOCKET_T);
```

In this call, the "`id`" parameter, `&hp->si`, is the pointer to a structure that will be passed to both the input function `wbs_io_in()` and the outputfunction `wbs_io_out()`.

Returns

The return code is a GIO error code defined in gio.h indicating the success or failure of the operation.

7 HTTP

There is just one function:

Name

`ht_get_form_XXX ()`

Syntax

```
int ht_get_form_XXX(struct httpd *hp, char *name, XXX *addr, int index)
```

Parameters

(see below)

Description

This function searches for a making name in the name/value array within the specified httpd structure. It writes the value of the variable at the address pointed to by the XXX parameter. If more than one name/value pair ...etc.

Returns

Number of name/value pairs with the specified name.

The following is a list of the general and the simpler version of the functions for returning values from a form:

```
/* Return a string */
int ht_get_form_ str(struct httpd *, char *name, char **str, int index)
char  *get_form_ str(struct httpd *hp, char *name);

/* Return an integer */
int    ht_get_form_int(struct httpd *hp, char *name, uint32_t *value, int index

int    get_form_int(struct httpd *hp , char *name, uint32_t *value);

/* Return a boolean-typically whether or not a checkbox or button was selected

int    ht_get_form_bool(struct httpd *hp, char *name, int *value, int index);

int    get_form_bool(struct httpd *hp, char *name);

/* Convert a dotten notation IP4 address into an ip_addr(4-byte binary address

int    ht_get_form_ip4addr(struct httpd *hp, char *name, ip_addr *ipptr, int in

char  *get_form_ip4addr(struct httpd *hp, char *name, ip_addr *ipptr);

/* Convert a colon separated ASCII IP6 address to a 16-byte binary address */

int    ht_get_form_ip6addr(struct httpd *hp, char *name, uint8_t *buf, int inde

int    get_form_ip6addr(struct httpd *hp, char *name, uint8_t *buf);
```

8 IKE

8.1 IkeAdminPrintLocalConf

Name

`IkeAdminPrintLocalConf()`

Syntax

```
int
IkeAdminPrintLocalConf (GIO *gio, char *policy_name)
```

Parameters

gio	pointer to the GIO context
policy_name	A simple alpha numeric string with no spaces used to reference the policy or NULL. When NULL information for all policies will be displayed.

Description

Displays all IKE information about a policy.

See also "ipsec netstat -p" to display ipsec table information about policies.

Returns

This function returns 0 on success, non-zero error code on failure.

8.2 IkeAdminPrintRemoteConf

Name

`IkeAdminPrintRemoteConf()`

Syntax

```
int
IkeAdminPrintRemoteConf (GIO *gio, char *remote_str)
```

Parameters

gio	pointer to the GIO context
remote_str	A simple alpha numeric string with no spaces used to reference the remote peer or NULL. When NULL information for all remote peers will be displayed.

Description

Displays all IKE information about a remote peer.

Returns

This function returns 0 on success, non-zero error code on failure.

8.3 ike_delete_ph1

Name

`ike_delete_ph1()`

Syntax

```
void
ike_delete_ph1(void)
```

Parameters

None

Description

Delete all IKEv1 Phase 1 SAs.

Returns

No value returned.

8.4 ike_delete_ph2

Name

`ike_delete_ph2()`

Syntax

```
void
ike_delete_ph2(void)
```

Parameters

None

Description

Delete all IKEv1 Phase 2 SAs.

Returns

No value returned.

8.5 ikev2_AddPolicy

Name

`ikev2_AddPolicy()`

Syntax

```
int
ikev2_AddPolicy (
        int          policy,
        Uchar        protocol,
        const char   *srcid_str,
        const char   *dstid_str,
        const char   *raddr_str,
        char         *auth_algs,
        char         *encr_algs,
        Uint32       sp_flags,
   Uint         new_ipsec_sa_lifetime,
        Uint         priority,
        Uint32       spid,
        char         *policy_name,
        char         *remote_name)
```

Parameters

policy	IPSEC_POLICY_SECURE_TRANSPORT or IPSEC_POLICY_SECURE_TUNNEL defined in ipsecapi.h
protocol	Upper layer (transport) protocol (e.g. TCP or UDP). Use 0 to specify any protocol.

srcid_str, dstid_str	Source and Destination Identities (Traffic Selectors). Each identity is a NULL terminated string in any of the following formats: • ipaddress • ipaddress/subnet_mask • ipaddress/subnet_bits • ipaddress_start-ipaddress_end • any • any6 In addition, an optional port number may be specified after the IP address or at the end of the string by comma (',') followed by port number. If a port number is used, protocol must either be IP_PROTO_UDP (17) or IP_PROTO_TCP (6). Port number cannot be specified with the any string. The following are some valid examples of identities: • "192.168.10.1" • "192.168.10.0/255.255.255.0" • "192.168.10.0/24" • "192.168.10.40-192.168.10.60" • "192.168.10.1,1500" • "192.168.10.0,1500/255.255.255.0" • "192.168.10.0/24,1500" • "192.168.10.40-192.168.10.60,1500" • "3ffe:501:ffff::211:11ff:febe:7f61" • "fe80::211:11ff:febe:7f61" • "3ffe:501:ffff:2::/64" • "any" • "any6" Note that 'any6' means any IPv6 address.
raddr_str	Remote IPSec endpoint's IP address in standard dotted notation.
encr_algs	Encryption algorithms. A comma separated list of the following: • 3DES • AES128 /* AES-CBC-128 from RFC3602 */ • AES192 /* AES-CBC-192 from RFC3602 */ • AES256 /* AES-CBC-256 from RFC3602 */

auth_algs	Authentication algorithms. A comma separated list of the following: • MD5 /* MD5 */ • SHA /* SHA-1 */ • SHA256 /* SHA2-256 from RFC4868 */ • SHA384 /* SHA2-384 from RFC4868 */ • SHA512 /* SHA2-512 from RFC4868 */
sp_flags	Policy Flags. Bitwise OR of any of the following: • SP_PROTO_AH • SP_PROTO_ESP • SP_CHECK_REPLAY • SP_CHECK_ESP_PAD • SP_FULL_ECN • SP_DF_COPY • SP_DF_SET • SP_KEYNEG_MANUAL • SP_TX_DUMMY_PKT • SP_TX_TFC_PADDING • SP_AUTH_SHA2_ICV_SHORT
ipsec sa lifetime	the valid lifetime of the SAs to be created:
priority	Policy priority. One of any of the following: • SP_PRIORITY_HIGHEST • SP_PRIORITY_HIGH • SP_PRIORITY_MED_HIGH • SP_PRIORITY_MEDIUM • SP_PRIORITY_MED_LOW • SP_PRIORITY_LOW • SP_PRIORITY_LOWEST
spid	The policy ID of the IPSEC policy database, returned by IPSecAdminAddPolicy
policy_name	A simple alpha numeric string with no spaces used to reference the policy.
remote_name	A simple alpha numeric string with no spaces used to reference the remote peer.

Description

Add a policy entry for IKE. A matching policy (and spid) must already exist in IPSEC.

See also IPSecAdminAddPolicy which adds policy information to the IPSEC database and returns the policy id, spid, used by IPSEC.

Returns

This function returns 0 on success, non-zero error code on failure.

8.6 ikev2CreateRemote

Name

`ikev2CreateRemote()`

Syntax

```
struct rcf_remote *
ikev2CreateRemote(char *remote_name,
        char   *encr_algs,        /* Encryption Alg */
        char   *auth_algs,            /* Auth Alg */
        char   *prf_algs,           /* PRF Alg */
        char   *auth_methods,           /* Auth Method Alg */
        char   *dh_groups,            /* Diffie-Hellman Alg */
        const char   *raddr_str,
        int accept_version,
        int initiate_version,
        int conf_version,
        int kmp_sa_lifetime_time,
        int interval_to_send,
        int times_per_send,
        char *local_certs,
        char *rem_cert,
        char *local_id,
        char *rem_id,
        char *psk,
   int  *err_code
        )
```

Parameters

remote_name	A simple alpha numeric string with no spaces used to reference the remote peer.
encr_algs	Encryption algorithms. A comma separated list of the following: • 3DES • AES128 /* AES-CBC-128 from RFC3602 */ • AES192 /* AES-CBC-192 from RFC3602 */ • AES256 /* AES-CBC-256 from RFC3602 */

auth_algs	Authentication algorithms. A comma separated list of the following: • MD5 /* MD5 */ • SHA /* SHA-1 */ • SHA256 /* SHA2-256 from RFC4868 */ • SHA384 /* SHA2-384 from RFC4868 */ • SHA512 /* SHA2-512 from RFC4868 */
prf_algs	Pseudo-Random Function, PRF, algorithms. A comma separated list of the following: • MD5 /* MD5 */ • SHA /* SHA-1 */ • SHA256 /* SHA2-256 from RFC4868 */ • SHA384 /* SHA2-384 from RFC4868 */ • SHA512 /* SHA2-512 from RFC4868 */
auth_methods	Authentication Method, one of the following: • PSK /* Pre-Shared Key */ • DSS /* Digital Signature */ • RSA /* X.509 Certificate */
dh_groups	Diffie-Hellman Group, a comma separated list of the following: • MODP768 • MODP1024 • MODP1536 • MODP2048 • MODP3072 • MODP4096 • MODP6144 • MODP3072
raddr_str	Remote IPSec endpoint's IP address in standard dotted notation.
accept_version	1 => accept IKEv1 requests from remote, 2 => accept IKEv2 Only applies when the same version is specified in the conf_version parameter. The values are OR'ed together so that both IKEv1 and IKEv2 can be accepted from the remote initiator

initiate_version	1 => initiate IKEv1 requests, 2 => initiate IKEv2 requests Only applies when the same version is specified in the conf_version parameter. Any other value makes the version being configured unable to initiate IKE SAs. Only one of IKEv1 or IKEv2 can be used to initiate IKE SAs.
conf_version	1 => configure IKEv1, 2 => configure IKEv2 Only one version can be configured with each call however multiple calls can be made to configure both IKEv1 and IKEv2.
kmp_sa_lifetime_time	SA lifetime
interval_to_send	interval_to_send -- retransmission interval
times_per_send	Not currently used.
local_certs	Documentation TBD
rem_cert	Documentation TBD
local_id	Documentation TBD
rem_id	Documentation TBD
psk	Documentation TBD

Description

Adds information about a remote peer for IKE.

Multiple calls to this function with the same remote_name are allowed because only IKEv1 or IKEv2 can be configured in s single call. All of the configuration information for IKEv1 and IKEv2 is indpendent so different algorithms could be selected for each version.

Returns

This function returns a point to a struct rcf_remote on success, NULL on failure.

8.7 ikev2_DeleteAllPolicies

Name

`ikev2_DeleteAllPolicies()`

Syntax

```
void
ikev2_DeleteAllPolicies(void)
```

Parameters

None

Description

Delete all policies from both IKE and IPSEC tables.

Returns

No return value.

8.8 ikev2_DeleteAllRemotes

Name

`ikev2_DeleteAllRemotes()`

Syntax

```
void
ikev2_DeleteAllRemotes(void)
```

Parameters

None

Description

Delete all remote peer information from IKE tables.

Returns

No return value.

8.9 ikev2_DeletePolicy

Name

`ikev2_DeletePolicy()`

Syntax

```
int
ikev2_DeletePolicy(char *policy_name)
```

Parameters

policy_name	A simple alpha numeric string with no spaces used to reference the policy.

Description

Delete a policy from both IKE and IPSEC tables.

Returns

This function returns 0 on success, non-zero error code on failure.

8.10 ikev2_DeleteRemote

Name

`ikev2_DeleteRemote()`

Syntax

```
void
ikev2_DeleteRemote(char *remote_name)
```

Parameters

remote_name	A simple alpha numeric string with no spaces used to reference the remote peer.

Description

Delete a remote peer information from IKE tables.

Returns

No value returned.

8.11 ikev2_shutdown

Name

`ikev2_shutdown()`

Syntax

```
void
ikev2_shutdown(void)
```

Parameters

None

Description

shut down all IKEv2 SAs by sending DELETE.

Returns

No value returned.

9 IP

9.1 add_route

API Name

`add_route()`

Syntax

RTMIB add_route(ip_addr dest, ip_addr mask, ip_addr nexthop, int iface, int prot);

Parameters

```
ip_addr dest /* ultimate destination */

ip_addr mask /* net mask, 0xFFFFFFFF if dest is host address */

ip_addr nexthop /* where to forward to */

int iface /* interface (net) for nexthop */

int prot /* how we know it: icmp, table, etc */
```

File

`ip/ip.c`

Description

Make an entry in the route table directing `dest` to `nexthop`.

Returns

Returns a pointer to the table entry; so caller can process it further, i.e. add metrics.

9.2 icmpEcho

API Name

`icmpEcho()`

Syntax

```
int icmpEcho(ip_addr host, unsigned length, unshort pingseq);
```

Parameters

host	host to ping - 32 bit, local-endian
length	total desired length of packet on media
pingseq	ping sequence number

File

`net/ping.c`

Description

Send an ICMP echo request (the guts of "ping"). Callable from Applications. Sends a single "ping" (ICMP echo request) to the specified `host`. The application must provide an appropriate `pingDemux()` routine if ping replies are to be checked.

Returns

Returns `0` if ping sent OK, else negative error code.

9.3 iproute

API Name

`iproute()`

Syntax

```
NET iproute(ip_addr host, ip_addr *hop1);
```

Parameters

```
ip_addr host /* IP address of final destination host */

ip_addr *hop1 /* IP address to use in resolving MAC address */
```

File

`ip/ip.c`

Description

Performs IP routing on an outgoing IP packet. Takes the Internet address to which we want to send a packet and returns the net interface through which to send it. An IP address is returned pointed to by the output parameter `hop1` which is the IP address for resolving the MAC destination address of the packets. If the target host is on our local segment, `hop1` will be the same as `host`, else it will be the IP address of the gateway or router through which we might be able to reach `host`.

Returns

Returns a pointer to a `net` structure which describes the interface of the MAC media we should send the packet on. Returns `NULL` when unable to route.

9.4 make_arp_entry

API Name

`make_arp_entry()`

Syntax

```
struct arptabent *make_arp_entry(ip_addr dest_ip, NET net);
```

Parameters

dest_ip	IP address to enter into table
net	Associated network interface

File

`ip/et_arp.c`

Description

Finds the first unused (or the oldest) ARP table entry and makes a new entry to prepare it for an ARP reply. If the IP address already has an ARP entry, the entry is returned with only the time stamp modified. The MAC address of the created entry is not resolved but left as zeros. The eventual ARP reply will fill in the MAC address.

Returns

Returns pointer to ARP table entry selected.

9.5 udp6_alloc

API Name

`udp6_alloc()`

Syntax

```
PACKET udp6_alloc(int datalen, int optlen, bool_t contig);
```

Parameters

datalen	length of UDP data (not including header)
optlen	length of IP options if any. Usually 0.
contig	Must use a single packet to hold of the headers and data

Description

This returns a `PACKET` big enough for the UDP data. It works by adding the space needed for UDP, IP, and MAC headers to the `datalen` passed and calling `pk_alloc()`. It also ensures that the `FREEQ_RESID` resource is locked around the call to `pk_alloc()`.

If the `contig` parameter is set, the call will fail if no free buffer queue contains a buffer large enough to hold the headers and all of the data. If `contig` is zero, the request can be satisfied by chaining packets together.

Returns

Returns a `PACKET` (pointer to struct `netbuf`) if OK, else `NULL` if a big enough packet was not available.

9.6 udp6_open

API Name

`udp6_open()`

Syntax

```
UDPCONN udp6_open(ip6_addr f6host,
                  unshort fsock,
                  unshort lsock,
                  int (*handler) (PACKET, void *, struct sockaddr *),
                  void * data);
```

Parameters

f6host	host to receive from, 'ip6unspecified' if any is OK
fsock	foreign socket (port) number, 0 if any is OK
lsock	local socket (port) to receive on
handler	callback function to be invoked upon receipt of application data
data	returned on upcalls to aid de-muxing

Description

This routine creates a structure in the UDP layer to receive and upcall UDP packets which match the parameter passed. The foreign host and socket can use `0` as a wild card. This allows us to start "listens" for incoming SNMP Stations, TFTP applications, etc.

The handler routine is passed three parameters:

1. A pointer to the struct netbuf data structure for the received packet, with nb_prot and nb_tlen set to point to the starting address and total length of the application data.
2. A copy of the 'data' parameter that was passed into udp6_open().
3. A pointer to the `struct sockaddr_in6` data structure containing the IPv6 source address and UDP port number of the sender.

Returns

Pointer to UDP Connection structure, or `NULL` on failure.

9.7 udp6_send

API Name

`udp6_send()`

Syntax

```
int udp6_send(ip6_addr * faddr. int scopeID, unshort fport, unshort lport,
PACKET p);
```

Parameters

faddr	Destination IPv6 address
scopeID	The scopeID for the destination address. This is only used on MULTI_HOMED systems when the destionation IP address is a link local address. This is a 1's based index of the egress interface.
fport	target UDP port
lport	local UDP port
p	packet to send, nb_prot ... nb_plen set to data, fhost set

Description

Send a UDP datagram to the foreign host in `p->fhost`. The '`local`' and '`remote`' ports in the UDP header are set from the values passed. Note: If `udp6_send()` is called without having first called `udp6_open()` on the associated port then any responses will be dropped.

Returns

`0` is OK, or a negative `ENP_` error code.

9.8 udp_alloc

API Name

`udp_alloc()`

Syntax

```
PACKET udp_alloc(int datalen, int optlen, bool_t contig);
```

Parameters

datalen	length of UDP data (not including udp header)
optlen	length of IP options if any. Usually 0.
contig	Must use a single packet to hold of the headers and data

File

`ip/udp.c`

Description

This returns a PACKET big enough for the UDP data. It works by adding the space needed for UDP, IP, and MAC headers to the `datalen` passed and calling `pk_alloc()`. It also ensures that the FREEQ_RESID resource is locked around the call to `pk_alloc()`.

If the contig parameter is set, the call will fail if no free buffer queue contains a buffer large enough to hold the headers and all of the data. If contig is zero, the request can be satisfied by chaining packets together.

Returns

Returns a PACKET (pointer to struct `netbuf`) if OK, else `NULL` if a big enough packet was not available.

9.9 udp_close

API Name

`udp_close()`

Syntax

```
void udp_close(UDPCONN con);
```

Parameters

```
UDPCONN con /* an open UDP connection */
```

File

`net/udp_open.c`

Description

`udp_close()` closes a udp connection, by removing the connection from UDP's list of connections and deallocating its internal structures.

Returns

Nothing.

9.10 udp_free

API Name

`udp_free()`

Syntax

```
void udp_free(PACKET p);
```

Parameters

p	ptr to netbuf structure previously allocated by udp_alloc()

File

`ip/udp.c`

Description

`udp_free()` is used to return a previously allocated `PACKET` to the InterNiche stack's free pool. It works by calling `pk_free()`, but like `udp_alloc()` it ensures that the `FREEQ_RESID` resource is locked around the access to the free packet pool.

Returns

Void.

9.11 udp_open

API Name

`udp_open()`

Syntax

```
UDPCONN udp_open(ip_addr fhost,
                 unshort fsock,
                 unshort lsock,
                 int (*handler) (PACKET, void *, struct sockaddr *),
                 void * data);
```

Parameters

fhost	host to receive from, 0 if any is OK
fsock	foreign socket (port) number, 0 if any is OK
lsock	local socket (port) to receive on
handler	udp received callback function
data	returned on upcalls to aid de-muxing

File

`ip/udp_open.c`

Description

This routine creates a structure in the UDP layer to receive and upcall UDP packets which match the parameter passed. The foreign host and socket can use `0` as a wild card. This allows us to start "listens" for incoming SNMP Stations, TFTP applications, etc.

The handler routine is passed three parameters:

1. A pointer to the struct netbuf data structure for the received packet, with nb_prot and nb_tlen set to point to the starting address and total length of the application data.
2. A copy of the 'data' parameter that was passed into udp_open().
3. A pointer to the struct sockaddr_in data structure containing the IPv4 source address and UDP port number of the sender.

Returns

Pointer to UDP Connection structure, or `NULL` on failure.

9.12 udp_send

API Name

`udp_send()`

Syntax

```
int udp_send(unshort fport, unshort lport, PACKET p);
```

Parameters

fport	target UDP port
lport	local UDP port
p	packet to send, nb_prot ... nb_plen set to data, fhost set

File

`ip/udp.c`

Description

Send a UDP datagram to the foreign host in `p->fhost`. The '`local`' and '`remote`' ports in the UDP header are set from the values passed. Note: If `udp_send()` is called without having first called `udp_open()` on the associated port then any responses will be dropped.

Returns

`0` is OK, or a negative `ENP_` error code.

10 IPSec

10.1 IPSecAdminAddBypassPolicy

API Name

`IPSecAdminAddBypassPolicy()`

Syntax

```
int IPSecAdminAddBypassPolicy(const char *srcid_str, const char *dstid_str,
Uchar protocol, Uint priority)
```

Parameters

srcid_str	Source Identity (traffic selector)
dstid_str	Destination Identity (traffic selector)
protocol	Upper layer (transport) protocol (e.g. TCP or UDP) (0 = any protocol)
priority	Policy priority

Description

This API is a wrapper function on top of IPSecMgmtAddPolicy() function and can be used instead of IPSecMgmtAddPolicy() for adding a policy to bypass IPsec processing on packets that match the specified traffic selectors. It takes NULL terminated ASCII strings for source and destination identities (traffic selectors).

Returns

This function returns 0 on success, non-zero error code on failure.

10.2 IPSecAdminAddDropPolicy

API Name

`IPSecAdminAddDropPolicy()`

Syntax

```
int IPSecAdminAddDropPolicy(const char *srcid_str, const char *dstid_str,
Uchar protocol, Uint priority)
```

Parameters

srcid_str	Source Identity (traffic selector)
dstid_str	Destination Identities (traffic selector)
protocol	Upper layer (transport) protocol (e.g. TCP or UDP) (0 = any protocol)
priority	Policy priority

Description

This API is a wrapper function on top of IPSecMgmtAddPolicy() function and can be used instead of IPSecMgmtAddPolicy() for adding a policy to drop packets that match the specified traffic selectors. It takes NULL terminated ASCII strings for source and destination identities (traffic selectors).

Returns

This function returns 0 on success, non-zero error code on failure.

10.3 IPSecAdminAddPolicy

API Name

`IPSecAdminAddPolicy()`

Syntax

```
int IPSecAdminAddPolicy(int policy, Uchar protocol, const char *srcid_str,
const char *dstid_str, const char *raddr_str, Uint32 flags, Uint priority,
Uint32 *spid)
```

Parameters

policy	Policy to apply (any one of IPSEC_POLICY_DROP, IPSEC_POLICY_BYPASS, IPSEC_POLICY_SECURE_TUNNEL, or IPSEC_POLICY_SECURE_TRANSPORT)
protocol	Upper layer (transport) protocol (e.g. TCP or UDP) (0 = any protocol)
srcid_str	Source Identity (traffic selector). Each identity is a NULL terminated string in any of the following formats: IP address, IP address/subnet mask, IP address/subnet bits, (start)IP address-(end)IP address, any, any6. Examples include "192.168.10.1", "192.168.10.0 /255.255.255.0", "192.168.10.0/24", "192.168.10.40-192.168.10.60", "192.168.10.1,1500", "192.168.10.0,1500/255.255.255.0", "192.168.10.0/24,1500", "192.168.10.40-192.168.10.60,1500", "3ffe:501:ffff::211:11ff:febe:7f61", "fe80::211:11ff: febe:7f61", "3ffe:501:ffff:2::/64", "any" (any IPv4 address), and "any6" (any IPv6 address).
dstid_str	Destination Identity (traffic selector)
raddr_str	Remote IPSec endpoint's IP address in standard notation. This can be NULL only if the policy is DROP or BYPASS.
flags	Policy flags (bitwise OR of any of the following: SP_PROTO_AH, SP_PROTO_ESP, SP_CHECK_REPLAY, SP_CHECK_ESP_PAD, SP_FULL_ECN, SP_DF_COPY, SP_DF_SET, SP_KEYNEG_MANUAL, SP_TX_DUMMY_PKT, SP_TX_TFC_PADDING, and SP_AUTH_SHA2_ICV_SHORT)
priority	Policy priority (any one of the following: SP_PRIORITY_HIGHEST, SP_PRIORITY_HIGH, SP_PRIORITY_MED_HIGH, SP_PRIORITY_MEDIUM, SP_PRIORITY_MED_LOW, SP_PRIORITY_LOW, or SP_PRIORITY_LOWEST)
spid	Pointer to integer that is used to store the policy identifier (output) (may be specified as NULL)

Description

This API is a wrapper function on top of IPSecMgmtAddPolicy function and can be used instead of IPSecMgmtAddPolicy for adding a policy. It takes NULL terminated ASCII strings for IP address and source and destination identities.

Returns

This function returns 0 on success, non-zero error code on failure.

10.4 IPSecMgmtAddPolicy

API Name

`IPSecMgmtAddPolicy()`

Syntax

```
int IPSecMgmtAddPolicy( int policy, const IPSecID *srcid, const IPSecID
*dstid, IPAddr *raddr, Uint32 flags, Uint priority, Uint32 *spid);
```

Parameters

policy	Policy to apply (any one of IPSEC_POLICY_DROP, IPSEC_POLICY_BYPASS, IPSEC_POLICY_SECURE_TUNNEL, or IPSEC_POLICY_SECURE_TRANSPORT).
srcid	Source Identity (traffic selector).
dstid	Destination Identity (traffic selector).
raddr	Remote IPSec endpoint's IP address. This can be NULL only if the policy is DROP or BYPASS.
flags	Policy flags (bitwise OR of any of the following: SP_PROTO_AH, SP_PROTO_ESP, SP_CHECK_REPLAY, SP_CHECK_ESP_PAD, SP_FULL_ECN, SP_DF_COPY, SP_DF_SET, SP_KEYNEG_MANUAL, SP_TX_DUMMY_PKT, SP_TX_TFC_PADDING, and SP_AUTH_SHA2_ICV_SHORT).
priority	Policy priority (any one of the following: SP_PRIORITY_HIGHEST, SP_PRIORITY_HIGH, SP_PRIORITY_MED_HIGH, SP_PRIORITY_MEDIUM, SP_PRIORITY_MED_LOW, SP_PRIORITY_LOW, or SP_PRIORITY_LOWEST)
spid	Pointer to integer that is used to store the policy identifier (output) (may be specified as NULL).

Description

This API is used for adding a policy into the IPsec security policy database (SPD). This API can be used either with manual keying (static security association (SA)) or with automated keying (IKE).

Returns

This function returns 0 on success, non-zero error code on failure.

10.5 IPSecMgmtAddSA

API Name

`IPSecMgmtAddSA()`

Syntax

```
int IPSecMgmtAddSA (const Uint32 spid, Uchar encr_alg, Uchar encr_keylen,
const Uchar *encr_i_key, const Uchar *encr_o_key, Uchar auth_alg, Uchar
auth_keylen, const Uchar *auth_i_key, const Uchar *auth_o_key, Uint32
esp_i_spi, Uint32 esp_o_spi, Uint32 ah_i_spi, Uint32 ah_o_spi);
```

Parameters

spid	Policy ID
encr_alg	Encryption algorithm (ALG_ESP_*)
encr_keylen	Encryption key length (bytes)
encr_i_key	Inbound encryption key
encr_o_key	Outbound encryption key
auth_alg	Authentication algorithm (ALG_ESP_HMAC_* or ALG_AH_*)
auth_keylen	Authentication key length (bytes)
auth_i_key	Inbound authentication key
auth_o_key	Outbound authentication key
esp_i_spi	ESP inbound SPI (host byte order)
esp_o_spi	ESP outbound SPI (host byte order)
ah_i_spi	AH inbound SPI (host byte order)
ah_o_spi	AH outbound SPI (host byte order)

Description

This is used to add a security association to SAD for manual keying of IPSec. This is only used if IKE is not used for key negotiation. The policy must be added before an SA can be added. If both AH and ESP protocols are used for a SA, the inbound SPIs must be same for both protocols.

Returns

This function returns 0 on success, non-zero error code on failure.

10.6 PacketDecapsulateSync

API Name

`PacketDecapsulateSync()`

Syntax

```
int PacketDecapsulateSync(Packet **pp, int domain, int *policyp)
```

Parameters

pp	Pointer to pointer to input packet (input), and pointer to pointer to output packet (output). The output packet can be one of the following: (1) encapsulated packet (when policy is SECURE), (2) input packet (when policy is BYPASS or DROP), or (3) NULL (an error occurred during processing)
domain	Domain on which this packet was received.
policyp	On successful completion, the policy applied to the packet. It is one of the following: (1) IPSEC_POLICY_BYPASS (IPSec processing was bypassed. The output packet is identical to the input packet. IP protocol processing must continue normally.), (2) IPSEC_POLICY_DROP (The packet must be dropped by the IP stack. The output packet is identical to the input packet.), (3) IPSEC_POLICY_SECURE_TUNNEL (The packet was secured in tunnel mode. The input packet has been replaced with the decapsulated packet. IP processing must be restarted on this packet for the encapsulated IP header.), or (4) IPSEC_POLICY_SECURE_TRANSPORT (The packet was secured in transport mode. The input packet has been replaced with the decapsulated packet.)

Description

This function is called to decapsulate a packet. The protocol field of the IP header is checked to see if the protocol is an IPSec protocol (AH or ESP). If the protocol field in the IP header of the packet indicates that it is an IPSec packet, then the packet is passed to the IPSec decapsulation module. If the protocol is not an IPSec protocol, then the packet is passed through the classification engine. If the policy for this packet indicates that the packet should have been an IPSec packet, then an error is returned.

Returns

This function returns 0 on success, non-zero error code on failure.

10.7 PacketEncapsulateSync

API Name

`PacketEncapsulateSync()`

Syntax

```
int PacketEncapsulateSync(Packet **pp, int domain, int *policyp)
```

Parameters

pp	Pointer to pointer to input packet (input), and pointer to pointer to output packet (output). The output packet can be one of the following: (1) encapsulated packet (when policy is SECURE), (2) input packet (when policy is BYPASS or DROP), or (3) NULL (an error occurred during processing)
domain	Domain on which this packet will be sent after IPSec.
policyp	On successful completion, the policy applied to the packet. It is one of the following: (1) IPSEC_POLICY_BYPASS (IPSec processing is bypassed. The output packet is identical to the input packet. IP protocol processing must continue normally.), (2) IPSEC_POLICY_DROP (The packet must be dropped by the IP stack. The output packet is identical to the input packet.), or (3a) IPSEC_POLICY_SECURE_TUNNEL / (3b) IPSEC_POLICY_SECURE_TRANSPORT (The packet has been secured. The input packet has been replaced with the encapsulated packet. The packet must be sent to the outbound network interface. IP must fragment the packet, if required, before sending it.)

Description

This function is called to encapsulate a packet. The packet is first passed through the classification engine. If the policy for this packet indicates that the packet must be secured, then it is passed to the IPSec engine and secured. Otherwise, it is returned back to the caller.

Returns

This function returns 0 on success, non-zero error code on failure.

10.8 PacketGetPolicy

API Name

`PacketGetPolicy()`

Syntax

```
int PacketGetPolicy(Packet *pkt, int domain, int *policyp, SP **spp)
```

Parameters

pp	Pointer to packet
domain	Domain identifier.
policyp	On successful completion, the policy that applies to the packet. It is one of the following: (1) IPSEC_POLICY_BYPASS (packet should bypass IPsec processing), (2) IPSEC_POLICY_DROP (packet should be dropped), (3) IPSEC_POLICY_SECURE_TUNNEL (packet should be secured in tunnel mode), or (4) IPSEC_POLICY_SECURE_TRANSPORT (packet should be secured in transport mode).
spp	Pointer to pointer to security policy structure. This pointer is filled in with the address of the matching security policy structure. (This parameter can be specified as NULL.)

Description

This function can be used to retrieve the policy that will be applied to the specified packet. The packet is classified, and if the classification is successful, the matching policy information is returned. The packet is not processed thru' IPsec. This function does not need to be explicitly called since the encapsulation process looks up the policy internally.

Returns

This function returns 0 on success, non-zero error code on failure.

11 Names and Addresses

11.1 dns_update

API Name

`dns_update()`

Syntax

```
dns_update(char *soa_mname, char *hname, struct sockaddr *ipaddr,
           int r_type, unsigned long ttl, void *pio)
```

Parameters

soa_mname	domain name
hname	host name
ipaddress	IPv4 or IPv6 address using the appropriate struct sockaddr. IPv6 link local addresses on MULTI_HOMED systems should include a scopeID.
int r_type	type of "A" record: 4 for IPv4 "A" or 6 for IPv6 "AAAA" record
long ttl	Time to live value
pio	GIO handle for output (or NULL)

Description

Sends a DNS UPDATE packet to the authoritative server with the specified domain name. First sends `DNS_TYPE_SOA` to get IP address of authoratative server. It then sends the `DNS_UPDT` packet to the authoritative server.

Returns

- 0 if successful
- Negative `ENP` error if internal error occurs (eg timeout)
- One of the `DNSRC_` errors from network (all positive).

11.2 freeaddrinfo

API Name

`freeaddrinfo()`

Syntax

```
void freeaddrinfo(struct addrinfo *ai);
```

Parameters

ai	Ptr to array of addrinfo structures returned by getaddrinfo()

Description

Frees the array of `addrinfo` structures returned by `getaddrinfo()`. It also frees the buffers within the structures that were used to hold names and addresses.

Returns

Nothing.

11.3 getaddrinfo

API Name

`getaddrinfo()`

Syntax

```
int getaddrinfo(CONST char *nodename, CONST char *servname,
                CONST struct addrinfo *hints, struct addrinfo **res);
```

Parameters

nodename	Domain name or an IP address
servname	Service name or port number
hints	Structure defined in RFC 3943
res	Ptr to array of 1 or more addrinfo structures.

Description

Translates a host name and/or service name and returns a set of socket addresses and associated info. to be used to create a socket to address the specified service with. This API is defined by RFC 3493 and intended to replace `gethostbyname()` and `gethostbyname2()`. It is thread safe. It is complex but provides many capabilities. It is available when DNSC_GETADDRINFO is defined.

The "hints" parameter is an `addrinfo` structure as defined in RFC 3943. On entry it contains a flags field, "`ai_flags`". The value in `ai_flags` is a hexidecimal `OR` of the desired "AI_" flags (`dns.h`). The flags direct the operation of the command and may limit the returned information.

The port number returned for a specified service name is based on `servtoportlist[ ]` in `dnsclnt.c`. The default array is limited in size. Add additional entries as needed for an implementation.

The function returns a pointer to an array of `addrinfo` structures with one structure for each address returned. On return, the calling application should use the info. in the structures as needed then call `freeaddrinfo()` to free the array.

Note

The `AI_V4MAPPED` flag is not currently supported, and the command does not currently support IP_V6 scope IDs other then 1

freeaddrinfo() must be called to free this array

Returns: 0 or one of the `EAI` error code defined in RFC_3493 and `dns.h`

11.4 gethostbyname

API Name

`gethostbyname()`

Syntax

```
struct hostent *gethostbyname(char *name);
```

Parameters

name	host name

Description

Get host information for named host. Implements a "standard" Unix version of gethostbyname(). Returns a pointer to a hostent structure if successful, NULL if not successful. The returned structure should NOT be freed by the caller.

Note

The returned hostent structure is not thread safe. It could be freed by internal DNS client routines if the entry ages out or if the table becomes full and space is needed for another entry.

Returns

Returns a pointer to host entry structure or NULL.

11.5 gethostbyname2

API Name

`gethostbyname2()`

Syntax

```
struct hostent *gethostbyname2(char *name, int af);
```

Parameters

name	host name
af	either `AF_INET` or `AF_INET6`

Description

Get host information for named host. Host information can be either in IPv4 or IPv6 format.

Note

Note: This API was deprecated by RFC 2553. The returned struct hostent has the same thread-safe problems described for gethostbyname().

Returns

Pointer to host entry structure or NULL

11.6 getnameinfo

API Name

`getnameinfo()`

Syntax

```
int getnameinfo(CONST struct sockaddr *sa, int salen, char *node,
                int nodelen, char *service, int servicelen, int flags)
```

Parameters

sa	Socket address. Either IPV4 or IP_V6
salen	Length of socket address
node	Buffer to contain the returned node name
nodelen	Length of buffer
service	IN: port number. OUT: Service name
servicelen	Length of service buffer
flags	Hexidecimal OR of desired NI_ flags (dns.h)

Description

Translates a socket address to a node name and/or a port number to a service name. The API behaves as defined in RFC 3493. The "`flags`" parameter can be used to change the default actons of the API. This API is available when `DNSC_GETADDRINFO` is defined.

Note

Note these NI_flags are NOT the same as the AI_flags for getaddrinfo().

Returns

0 or one of the `EAI` error code defined in RFC_3493 and `dns.h`

11.7 in46_reshost

API Name

`in46_reshost()`

Syntax

```
int in46_reshost(char *host, int type, struct dns_querys *ret_DNS_Entry,
int flags);
```

Parameters

host	Host name string or IP address string
type	DNS record type (see dns.h)
ret_DNS_Entry	In: allocated buffer. Out: copy of a DNS entry
flags	RH_VERBOSE, RH_BLOCK

Description

Mid-level thread-safe API used to resolve a host name to an IP address or to obtain a host name for an IP address. The DNS cache will be searched first for the requested information. If the information is not available there, it will make calls to the DNS servers. If `RH_BLOCK` is set the call will not return until the address is resolved or a timeout occurs. When called, `ret_DNS_Entry` must contain a buffer large enough to hold a `dns_querys` structure. When the function returns 0 (success), the "`ret_DNS_entry` " will contain a copy of a DNS entry. This buffer must be freed by the application.

Returns

0 if address was set, else one of the `ENP_` error codes

11.8 inet_ntop

Name

`inet_ntop()`

Syntax

```
const char *inet_ntop(int af, const void *addr, char *str, size_t size);
```

Parameters

af	Address family (AF_INET or AF_INET6)
addr	Pointer to storage for IPv4 address ('ip_addr') or IPv6 address ('struct in6_addr') in network byte order
str	Pointer to storage for string that will contain IPv4 address in dotted decimal notation, or an IPv6 address in colon-separated notation
size	Length of output buffer ('str')

Description

This functions converts a binary representation of an IPv4 address or IPv6 address (in network byte order) into a string in dotted decimal notation. The output buffer must be at least 16 (or 40) bytes long for an IPv4 (or IPv6) address.

Returns

This function returns NULL if it encountered an error; otherwise, it returns the third argument ('str').

11.9 inet_pton

API Name

`inet_pton()`

Syntax

```
int inet_pton(int af, const char *src, void *dst);
```

Parameters

af	Address family (AF_INET or AF_INET6)
src	Pointer to string containing IPv4 address in dotted decimal notation, or an IPv6 address in colon-separated notation
dst	Pointer to storage for IPv4 address ('ip_addr') or IPv6 address ('struct in6_addr') where the results of the conversion will be stored (in network byte order)

Description

This functions converts a string containing an IPv4 or IPv6 address in printable format into its equivalent binary representation (in network byte order).

Returns

This function returns 0 if the conversion was successful. A non-zero return value indicates a failure.

11.10 ip_mymach

API Name

`ip_mymach()`

Syntax

```
ip_addr ip_mymach(ip_addr host);
```

Parameters

```
ip_addr host /* IP address of foreign host to find */
```

File

`ip/ip.c`

Description

Returns the address of our machine relative to a given foreign host IP address. On a single homed host this will always return the sole interface's IP address; on a router it will return the address of the interface to which packets for the host would be routed.

Returns

Our IP address on one of our networks interfaces.

11.11 nslookupr

API Name

`nslookupr()`

Syntax

```
int nslookupr(char *name, char type, struct dns_querys **dns_entry);
```

Parameters

char *name	name to lookup
char type	lookup type (must be DNS_TYPE_PTR)
dns_querys **dns_entry	ptr to DNS query structure

Description

Performs a reverse lookup

Returns

Pointer to host entry structure or NULL

11.12 parse_ipad

API Name

`parse_ipad()`

Syntax

```
char * parse_ipad(ip_addr * ipout, unsigned * sbits, char * stringin);
```

Parameters

ipout	pointer to IP address to set
sbits	default subnet bit number
stringin	buffer with ascii to parse

File

`misclib/parseip.c`

Description

Looks for an IP address in `stringin` buffer, makes an IP address (in big-endian) in `ipout`.

Returns

Returns `NULL` upon success, else returns a pointer to a string describing the syntax problem in the input string.

11.13 print_ipad

API Name

`print_ipad()`

Syntax

```
char *print_ipad(unsigned long ipaddr);
```

Parameters

```
unsigned long ipaddr /* IP address to print, in Big-Endian (net order) */
```

File

`misclib/in_utils.c`

Description

Accepts a 32 bit IP address in big-endian format and returns a pointer to a volatile buffer with a printable version of the address. The buffer will be overwritten by each subsequent call to `print_ipad`, so the caller should copy it or use it immediately.

Note that the current implementation of `print_ipad()` is not re-entrant, and should not be used on a port to a pre-emptive RTOS.

Returns

Returns a pointer to the buffer with the printable IP address text.

12 Packets

12.1 pk_copy

API Name

`pk_copy()`

Syntax

```
PACKET pk_copy(char * in, int len, int disp);
```

Parameters

in	Pointer to source buffer
len	Number of bytes to be copied
disp	Displacement. Number of bytes to be left in front of the data

Description

pk_copy () allocates a packet chain and copies the specifed amount of data from the source buffer to the newly allocated buffer chain. The nb_prot pointer is set "disp" bytes from nb_buff, and disp bytes are added to the length of the first buffer. The disp value may be negative. Implicitly, MaxLnh is adjusted according to disp.

Returns

Pointer to the newly allocated PACKET or NULL.

12.2 pk_free

API Name

`pk_free()`

Syntax

```
void pk_free(PACKET pkt);
```

Parameters

PACKET pkt	Pointer to netbuf structure previously allocated by pk_alloc()

File

`net/pktalloc.c`

Description

`pk_free()` is used to return a previously allocated `netbuf` structure to the pool of such structures that is maintained by the InterNiche stack. When a packet buffer is passed to NicheStack, it is the responsibility of the lowest layer that handles the packet to free it. Normally, this would be the interface driver. However, it may be another layer such as TCP or IP if, because of a timeout or error, it does not pass the packet to a lower layer. The porting engineer should include a call to `pk_free()` in his network interface code in order to return a `netbuf` structure and its associated packet buffer to the free pool after the packet has been transmitted by the network device. For a description of how this is performed, see the description of pkt_send.

Note that if you happen to be implementing Mutual Exclusion using the Net Resource Method, then the FREEQ_RESID resource would need to be locked and unlocked while making calls to `pk_free()`.

Returns

Nothing.

12.3 pk_gather

API Name

`pk_gather()`

Syntax

```
PACKET pk_gather(PACKET pkt, int headerlen);
```

Parameters

pkt	Pointer to first packet in chain to be coalesced
headerlen	Displacement from nb_buff to nb_prot in the output packet

Description

`pk_gather()` allocates a single packet with a buffer long enough to hold the entire input packet chain. It is typically used when it is necessary to gather a chain into single buffer, for user processing or to pass to a driver that does not do gather.

Note: `pk_gather` uses `npalloc()` to allocate the packet and the buffer, and it sets the flag `PKF_COALESCED`. When `PKF_COALESCED` is set, the interface code must free the packet using `npfree()`, rather than putting the packet back on the free queue.

Returns

Pointer to newly allocated PACKET or NULL.

12.4 pk_init

API Name

`pk_init()`

Syntax

```
int pk_init(int len, int num);
```

Parameters

len	Length in bytes to be allocated.
num	Number of buffers to allocate.

Description

pk_init() allocates num buffers of length len in a free buffer queue. It will typically be called several times to allocate the various queues of buffers needed by the port. The function will fail if:

- len < 0
- too small: len < (6 * MaxLnh)
- too large: len > MAXCHAINDPKTSZ,
- too many packets : num > MAXCHAINDPKTNUM.
- An entry of that length has already been entered.
- There are no more free slots in the array buffer queues.
- Allocation of buffers has failed for lack of RAM.

Returns

0 for success or -1 if an error occurred.

12.5 pkt_send

API Name

`pkt_send() - Insert frame into network driver queue`

Syntax

```
int pkt_send(PACKET pkt);
```

Parameters

```
typedef struct netbuf *PACKET

PACKET pkt /* pointer to netbuf structure containing frame to send */
```

Description

This routine is responsible for sending the data described by the passed `pkt` parameter and queuing the `pkt` parameter for later release by the device driver. If the MAC hardware is idle the actual transmission of the packet should be started by this routine, else it should be scheduled to be sent later (usually by an "end of transmit" interrupt (EOT) from the hardware).

The PACKET type is described in the section titled **The netbuf Structure and the Packet Queues**". All the information needed to send the packet is filled into the structure addressed by this type before this call is made. Some of the important fields are:

```
pkt->nb_prot;   /* pointer to data to send. */
pkt->nb_plen;   /* length of data to send */
pkt->net;      /* nets[ ]structure for posting statistics */
```

The data addressed by `pkt->nb_prot` may or may not have already been prefixed with a MAC layer header depending on how the `nets[ ]` structure associated with the interface (`pkt->net`) has been configured. The rule for determining whether the MAC layer header is present or not can be expressed with the following pseudocode fragment.

```
if ((pkt->net->n_mibifType == SLIP) || (pkt->net->n_mib->ifType == PPP) || (pkt

        0)) the packet at pkt->nb_prot is not encapsulated with a MAC header; e

        is encapsulated with a MAC header;
```

If the if statement in the above pseudocode evaluates to TRUE then the packet at nb_prot is not encapsulated with a MAC header and it is up to the network interface code to transmit the MAC header that is appropriate for the network medium (if any). On the other hand, if the "if" statement evaluates to FALSE then appropriate MAC headers for media such as Ethernet or Token Ring will have been placed at the head of the buffer passed by the calling routine and are not the responsibility of this routine; however some drivers may have to access, strip or modify the MAC header if they are layered on top of complex lower layers. The ODI pkt_send() routine is an example of this (see doslib/odi.c).

Regardless of whether it is the responsibility of the network interface layer to transmit the MAC header, it is necessary for the network interface to transmit the nb_plen bytes starting at nb_prot plus "any" MAC header bias that was used to align the start of the IP header. For Ethernet devices, the macro ETHHDR_BIAS is sometimes defined to 2 bytes, to align the IP header at a 4 byte boundary. Likewise, the number of bytes to transmit in this case would be (nb_plen - ETHHDR_BIAS), if ETHHDR_BIAS was defined to non-zero. When all the bytes are sent, the structure addressed by the PACKET type should be returned to the free queue by a call to pk_free(), which may be called at interrupt time. Do not free the packet before it has been entirely sent by the hardware, since it may be reused (and its buffer altered) by the IP stack.

The simplest way to implement this routine is to block (busy-wait) until the data is sent. This allows for fast prototyping of new drivers, but will generally hurt performance. The usual design followed by InterNiche in the example drivers is to put the packet in an awaiting_send queue, check to see if the hardware is idle, and then call a send_next_from_q routine to dequeue the packet at the head of the send queue and begin sending it. The "end of transmit" ISR (EOT) frees the just sent packet and again calls the send_next_from_q routine. By moving all the PACKETs through the awaiting_send queue we ensure that they are sent in FIFO order, which significantly improves TCP and application performance.

If your hardware (or lower layer driver) does not have an end of transmit (EOT) interrupt or any analogous mechanism, you may need to use the raw_send() alternative to this function.

Slow devices (such as serial links), and hardware which DMAs data directly out of predefined memory areas, may copy the passed buffer into driver managed memory buffers, free the PACKET and return immediately; however they should be prepared to be called with more packets before transmission is complete.

Interface transmit routines should also maintain system statistics about packet transmissions. These are kept in the IfMib structure that is addressed by the n_mib field in each nets[ ] entry. Exact definitions of all these counters are available in RFC1213. At a minimum you should maintain packet byte and error counts since these can aid greatly with debugging your product during development and isolating configuration problems in field. Statistics keeping is best done at EOT time, but can be approximated in this call. The following fragment of code is a generic example:

```
/* compile statistics about completed transmit */
eth = (struct ethhdr *)pkt->nb_prot;   /* get ether header */
ifc = pkt->net;
if(send_status == SUCCESSFUL)   /* send_status set by hardware EOT */
{
    if(eth->e_dst[0] ... 0x01)      /* see if multicast bit is on */
        ifc->n_mib->ifOutNUcastPkts++;
     else
        ifc->n_mib->ifOutUcastPkts++;

    ifc->n_mib->ifOutOctets +=pkt->nb_plen;
}
else   /* error sending packet */
{
    ifc->n_mib->ifOutErrors++;
}
```

Returns

Returns `0` if OK, else one of the `ENP_` codes. Since this routine may not be waiting for the packet transmission to complete, it is permissible to return a 0 if the packet has been successfully queued for send or the send is in progress. Error (non-zero) codes should only be returned if a distinct hardware (or lower layer) failure is detected. There is no mechanism to report errors detected in previous packets or during the EOT. Upper layers like TCP will retry the packet when it is not acknowledged.

See Also

raw_send

12.6 raw_send

API Name

`raw_send()`

Syntax

```
int raw_send(NET net, char * data, unsigned data_bytes);
```

Parameters

```
typedef struct net *NET

NET net /* pointer to net structure to send it on */

char *data /* pointer to data buffer to send */

unsigned data_bytes /* number of bytes to send (length of data) */
```

Description

This routine should transmit the data as indicated on the device corresponding to the `net` parameter passed. A MAC header may or may not have been prefixed to the IP data depending on how the `nets [ ]` structure addressed by the net parameter has been configured. See the description of MAC headers in the description of the pkt_send function. This routine should not return until it is through with the data in the passed buffer, as the buffer may be reused (thus corrupting the data) immediately upon return.

The `pkt_send()` routine should be used instead of this one if there is an end of transmit interrupt (EOT) available on the hardware. This routine was designed for old DOS "packet driver" specification drivers which did not support EOT and should generally not be used on modern designs.

Slow devices (such as serial links), and hardware which DMAs data directly out of predefined memory areas may copy the passed buffer into driver managed memory and return immediately; however they should be prepared to be called with more buffers before transmission is complete.

Interface transmit routines should also maintain system statistics about packet transmissions. These are kept in the `n_mib` structure attached to each `nets[ ]` entry. Exact definitions of all these counters are available in RFC1213. At a minimum you should maintain packet byte and error counts since these can aid greatly with debugging your product during development and isolating configuration problems in the field. Statistics keeping is best done at EOT time, but can be approximated in this call. The following fragment of code is an example that works for Ethernet devices:

```
/* compile statistics about completed transmit */
eth = (struct ethhdr *)pkt->nb_prot;   /* get ether header */
if(send_status == SUCCESSFUL)   /* send_status set by hardware EOT */
{
   if(eth->e_dst[0] ... 0x01)         /* see if multicast bit is on */
      ifc->n_mib->ifOutNUcastPkts++;
   else
      ifc->n_mib->ifOutUcastPkts++;

   ifc->n_mib->ifOutOctets +=pkt->nb_plen;
}
else         /* error sending packet */
{
   ifc->n_mib->ifOutErrors++;
}
```

Returns

Returns `0` if OK, else one of the `ENP_` codes.

See Also **pkt_send()**

13 Sockets

13.1 t_accept

Syntax

```
long t_accept(long s, struct sockaddr *addr, int *addrlen);
```

Parameters

s	Socket identifier
addr	Pointer to struct sockaddr_in structure containing addressing information for the remote end in newly accepted connection
addrlen	Pointer to storage for length of sockaddr_in structure (bytes)

Description

The argument `s` is a socket that has been created with `t_socket()`, bound to an address with `t_bind()` and is listening for connections after a `t_listen()`. `t_accept()` extracts the first connection on the queue of pending connections, creates a new socket with the same properties as `s` and allocates a new file descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, `t_accept()` blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, `t_accept()` returns an error as described below. The accepted socket is used to read and write data to and from the socket which connected to this one; it is not used to accept more connections. The original socket s remains open for accepting further connections.

The argument `addr` is a result parameter that is filled in with the address of the connecting entity as known to the communications layer, i.e. the exact format of the `addr` parameter is determined by the domain in which the communication is occurring. The `addrlen` is a value-result parameter. It should initially contain the amount of space pointed to by `addr`. On return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with `SOCK_STREAM`.

It is possible to `t_select()` a socket for the purposes of doing an `t_accept()` by selecting it for read.

Returns

`t_accept()` returns a non-negative descriptor for the accepted socket on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also**: t_bind(), t_connect(), t_listen(), t_select(), t_socket()**

13.2 t_bind

API Name

`t_bind()`

Syntax

```
int t_bind(long , struct sockaddr *name, int namelen);
```

Parameters

s	Socket identifier
name	Pointer to struct sockaddr_in structure containing addressing information for local end
namelen	Length of sockaddr_in structure (bytes)

Description

`t_bind()` assigns a name to an unnamed socket. When a socket is created with `t_socket()` it exists in a name space (address family) but has no name assigned. `t_bind()` requests that the name pointed to by name be assigned to the socket.

Returns

`t_bind()` returns `0` on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_connect(), t_getsockname(), t_listen(), t_socket()

13.3 t_connect

API Name

`t_connect()`

Syntax

```
int t_connect(long s, struct sockaddr *name, int namelen);
```

Parameters

s	Socket identifier
name	Pointer to struct sockaddr_in structure containing addressing information for remote end (peer). IPv6 link local addresses on MULTI_HOMED systems should include a scopeID.
namelen	Length of sockaddr_in structure (bytes)

Description

The parameter `s` is a socket. If it is of type `SOCK_DGRAM` or `SOCK_RAW`, then this call specifies the peer with which the socket is to be associated; the address to which datagrams are sent and the only address from which datagrams are received. If it is of type `SOCK_STREAM`, then this call attempts to make a connection to another socket. The other socket is specified by `name` which is an address in the communications space of the socket. Each communications space interprets the `name` parameter in its own way.

Generally, stream sockets may successfully call `t_connect()` only once, however in the Interniche Sockets implementation even for a streams socket, if `NB_CONNECT` is defined in `ipport.h` and the socket is a non-blocking socket, then a socket allows repeated calls to `t_connect()`. These calls will return `0` once the socket is connected, or a `SOCKET_ERROR` if it is in the process of connecting.

Datagram and raw sockets may use `t_connect()` multiple times to change their association. Datagram and raw sockets may also dissolve the association by connecting to an invalid address, such as a zero address.

Returns

This returns `0` on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_accept(), t_connect(), t_getsockname(), t_select(), t_socket()

13.4 t_errno

Name

`t_errno ()`

Syntax

```
int t_errno(long s);
```

Parameters

s	Socket identifier

Description

This function returns the socket error associated with the specified socket.

Returns

This function returns ENOTSOCK if the socket identifier is not valid. Otherwise, it returns the socket error associated with the specified socket.

13.5 t_getpeername

API Name

`t_getpeername()`

Syntax

```
int t_getpeername(long s, struct sockaddr *name, int * addrlen);
```

Parameters

s	Socket identifier
name	Pointer to struct sockaddr_in structure containing addressing information for local end
addrlen	Pointer to storage for length of sockaddr_in structure (bytes)

Description

Fills in the passed `struct sockaddr` with the IP addressing information of the connected host.

Returns

This returns `0` on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_bind(), t_socket()

13.6 t_getsockname

API Name

`t_getsockname()`

Syntax

```
int t_getsockname(long s, struct sockaddr *name, int * addrlen);
```

Parameters

s	Socket identifier
name	Pointer to struct sockaddr_in structure containing addressing information for local end
addrlen	Pointer to storage for length of sockaddr_in structure (bytes)

Description

`t_getsockname()` returns the current name for the specified socket, in the passed `struct sockaddr`.

Returns

This returns `0` on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_bind(), t_getpeername(), t_socket()

13.7 t_getsockopt, t_setsockopt

API Name

`t_getsockopt()`

`t_setsockopt()`

Syntax

```
int t_getsockopt(long s, int level, int optname, char *optval, int optlen);

int t_setsockopt(long s, int level, int optname, char *optval, int optlen);
```

Parameters

s	Socket identifier
level	Level of socket option (IP_OPTIONS or SOL_SOCKET)
optname	Name of socket option (e.g., SO_ERROR)
optval	Pointer to storage for socket option (for reading (get) or writing (set))
optlen	Size of storage pointed to by 'optval'

Description

`t_getsockopt()` and `t_setsockopt()` manipulate options associated with a socket. The `optname` parameter identifies an option that is to be set with `t_setsockopt()` or retrieved with `t_getsockopt()`.

The parameter `optval` is used to specify option values for `t_setsockopt()`. On calls to `t_setsockopt()` it generally contains a pointer to a variable or structure, the contents of which will define the value of the option to be set. On calls to `t_getsockopt()` it generally points to a variable or structure into which the value for the requested option is to be returned.

The include file `socket.h` contains definitions for option names, described below. Most options take a pointer to an int variable for `optval`. For `t_setsockopt()`, the variable addressed by the parameter should be non-zero to enable a Boolean option or zero if the option is to be disabled.

`SO_LINGER` uses a `struct linger` parameter defined in `socket.h`. This parameter specifies the desired state of the option and the linger interval (see below).

In addition to those referenced in Quick List for Socket Options, the following options are recognized by the InterNiche stack. Except as noted, each may be examined with `t_getsockopt()` and set with `t_setsockopt()`.

SO_DEBUG	toggle recording of debugging information
SO_REUSEADDR	toggle local address reuse
SO_KEEPALIVE	toggle keep connections alive
SO_DONTROUTE	toggle routing bypass for outgoing messages
SO_LINGER	linger on close if data present
SO_BROADCAST	toggle permission to transmit broadcast messages
SO_OOBINLINE	toggle reception of out-of-band data in band
SO_SNDBUF	set buffer size for output
SO_RCVBUF	set buffer size for input
SO_TYPE	get the type of the socket (get only)
SO_ERROR	get and clear error on the socket (get only)
SO_CALLBACK	set a callback function for the socket (set only)
IP_HDRINCL	set inclusion of IP header in data (SOCK_RAW only)

SO_DEBUG enables debugging in the underlying protocol modules.

SO_REUSEADDR indicates that the rules used in validating addresses supplied in a t_bind() call should allow reuse of local addresses.

SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken. If the process is waiting in t_select() when the connection is broken, t_select() returns true for any read or write events selected for the socket.

SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when unsent messages are queued on socket and a t_socketclose() is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the caller on the t_socketclose() attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the t_setsockopt() call when SO_LINGER is requested). If SO_LINGER is disabled and a t_socketclose() is issued, the system will process the close in a manner that allows the caller to continue as quickly as possible.

The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. Note that the InterNiche stack supports the setting and getting of this option for compatibility but does not check its value when transmitting broadcast messages.

With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received. It will then be accessible with t_recv() calls without the MSG_OOB flag.

SO_SNDBUF and SO_RCVBUF are options to adjust the normal buffer sizes allocated for the output and input buffers respectively. The buffer size may be increased for high-volume connections or may be decreased to limit possible backlog of incoming data. The system places an absolute limit on the values.

SO_TYPE and SO_ERROR are options used only with t_getsockopt(). SO_TYPE returns the type of the socket, for example SOCK_STREAM. SO_ERROR returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

IP_HDRINCL option is used only with SOCK_RAW sockets. The option value (pointed to by optval) is expected to be an integer; if it is non-zero it allows application access to the IP header, meaning that received datagrams include an IP header and sent datagrams are expected to be constructed with an IP header at the start of the buffer passed to the t_send() function. Its default setting is 0.

The options SO_NONBLOCK, SO_NBIO, and SO_BIO are unique to the InterNiche stack (these options do not appear in the Berkeley Sockets API) and are used to control whether a socket uses blocking or non-blocking IO.

SO_NONBLOCK allows the caller to specify blocking or non-blocking IO that works the same as the other Boolean socket options. That is, optval points to an integer value which will contain a non-zero value to set non-blocking IO or a 0 value to reset non-blocking IO. This means that we can get the current blocking or non-blocking status of a socket with t_getsockopt().

For compatibility, older InterNiche Sockets options SO_NBIO and SO_BIO are still supported. SO_NBIO is used to specify that a socket use non-blocking IO. SO_BIO is used to specify that a socket use blocking IO. The use of t_setsockopt() to set these options is different than that of the standard Boolean options in that the value in optval is not used. All that is necessary is to specify the appropriate option name in optname.

SO_NBIO	Set socket to use non-blocking IO.
SO_BIO	Set socket to use blocking IO.

The SO_CALLBACK option is also specific to the InterNiche stack and is only available if the stack has been built with the TCP_ZEROCOPY option enabled.

Returns

These return 0 on success. On failure, they return -1 and set an internal t_errno to one of the errors listed in Sockets Errors to indicate the error. The t_errno can be retrieved by a call to t_errno(s).

See Also

Quick List for Socket Options

t_socket()

13.8 t_listen

API Name

`t_listen()`

Syntax

```
int t_listen(long s, int backlog);
```

Parameters

s	Socket identifier
backlog	Used to compute a limit on the maximum number of connections that can be pending in the completed (those for which the TCP three-way handshake has completed) and partially completed (those for which the TCP three-way handshake has started, but isn't complete) queues.

Description

To accept connections, a `socket` is first created with `t_socket()`, a backlog for incoming connections is specified with `t_listen()` and then the connections are accepted with `t_accept()`. The `t_listen()` call applies only to sockets of type `SOCK_STREAM`. The backlog parameter defines the maximum length for the queue of pending connections (not maximum open connections). If a connection request arrives with the queue full the client will receive an error with an indication of `ECONNREFUSED`.

Returns

Returns `0` on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_accept(), t_connect()

13.9 t_recv, t_recvfrom

API Name

`t_recv()`

`t_recvfrom()`

Syntax

```
int t_recv(long s, char * buf, int len, int flags);
```

```
int t_recvfrom(long s, char *buf, int len, int flags, struct sockaddr
*from, int *fromlen);
```

Parameters

s	Socket identifier
buf	Start address of buffer where received data will be copied into
len	Length of data to be sent
flags	Flags for receiving process (e.g., MSG_PEEK)
from	Pointer to struct sockaddr_in structure containing addressing information for the remote end
fromlen	Pointer to storage for length of sockaddr_in structure (bytes)

Description

`s` is a socket created with `t_socket()`. `t_recv()` and `t_recvfrom()` are used to receive messages from another socket. `t_recv()` may be used only on a connected socket (see t_connect), while `t_recvfrom()` may be used to receive data on a socket whether it is in a connected state or not.

If `from` is not a `NULL` pointer, the source address of the message is filled in. `fromlen` is a value-result parameter, initialized to the size of the buffer associated with `from`, and modified on return to indicate the actual size of the address stored there. The length of the message is returned. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see t_socket).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is non-blocking (see t_setsockopt) in which case `-1` is returned with the external variable `t_errno` set to `EWOULDBLOCK`.

Note that `t_recv()` will return an `EPIPE` if an attempt is made to read from an unconnected socket.

The `t_select()` call may be used to determine when more data arrive.

The `flags` parameter is formed by OR-ing one or more of the following:

MSG_OOB	Read any "out-of-band" data present on the socket, rather than the regular "in-band" data.
MSG_PEEK	"Peek" at the data present on the socket; the data are returned, but not consumed, so that a subsequent receive operation will see the same data.

Returns

These calls return the number of bytes received, or `-1` if an error occurred. On failure, they set an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_connect(),t_getsockopt(),t_select(), t_send(), t_socket()

13.10 t_select

Syntax

```
int t_select (fd_set * readfds, fd_set * writefds, fd_set * exceptfds, long
tv);

void FD_SET (long so, fd_set * set)

void FD_CLR (long so, fd_set * set)

void FD_ISSET (long so, fd_set * set)

void FD_ZERO (fd_set * set)
```

Parameters

s	Socket identifier
readfds	Set of descriptors that an application will wait to become ready for reading
writefds	Set of descriptors that an application will wait to become ready for writing
exceptfds	Set of descriptors that an application will wait for occurence of an exceptional condition on
tv	Timeout duration (ticks)

Description

`t_select()` examines the socket descriptor sets whose addresses are passed in `readfds`, `writefds`, and `exceptfds` to see if some of their descriptors are ready for reading, ready for writing or have an exception condition pending. On return, `t_select()` replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned. Any of `readfds`, `writefds`, and `exceptfds` may be given as `NULL` pointers if no descriptors are of interest. Selecting true for reading on a socket descriptor upon which a `t_listen()` call has been performed indicates that a subsequent `t_accept()` call on that descriptor will not block.

In the standard Berkeley UNIX Sockets API, the descriptor sets are stored as bit fields in arrays of integers. This works in the UNIX environment because under UNIX socket descriptors are file system descriptors which are guaranteed to be small integers that can be used as indexes into the bit fields.

In the InterNiche stack, socket descriptor are pointers and thus a bit field representation of the descriptor sets is not feasible. Because of this, the InterNiche Sockets API differs from the Berkeley standard in that the descriptor sets are represented as instances of the following structure:

```
typedef struct fd_set {      /* the select socket array manager */
   unsigned fd_count;      /* how many are SET? */
   long fd_array[FD_SETSIZE];   /* an array of SOCKETs */
} fd_set;
```

Instead of a socket descriptor being represented in a descriptor set via an indexed bit, an InterNiche socket descriptor is represented in a descriptor set by its presence in the `fd_array` field of the associated `fd_set` structure. Despite this non-standard representation of the descriptor sets themselves, the following standard entry points are provided for manipulating such descriptor sets: `FD_ZERO(&fdset)` initializes a descriptor set `fdset` to the null set. `FD_SET(fd, &fdset)` includes a particular descriptor, `fd`, in `fdset`. `FD_CLR(fd, &fdset)` removes `fd` from `fdset`. `FD_ISSET (fd, &fdset)` is nonzero if `fd` is a member of `fdset`, zero otherwise. These entry points behave according to the standard Berkeley semantics.

The porting engineer should be aware that the value of `FD_SETSIZE` defines the maximum number of descriptors that can be represented in a single descriptor set. The default value of `FD_SETSIZE` of 12 is defined in `tcp/tcpport.h`. This value can be increased to accommodate a larger maximum number of descriptors at the cost of increased processor stack usage.

Another difference between Berkeley and InterNiche `t_select()` calls is the representation of the timeout. Under Berkeley, the timeout parameter is represented by a pointer to a structure. Under InterNiche Sockets, a timeout is specified by the `tv` parameter, which defines the maximum number of ticks that should elapse before the call to `t_select()` returns. A tv parameter equal to `0` implies that `t_select()` should return immediately (effectively a poll of the sockets in the descriptor sets). Note that there is no provision for no timeout, that is, there is no way to specify that `t_select()` block forever unless one of its descriptors becomes ready. The maximum value (longest time in ticks) that can be specified for the `tv` parameter can be calculated by dividing the largest value that can be represented in a variable of type `long` by the TPS constant (system ticks per second). On PC based systems where longs are typically 32 bits and TPS is 20, this works out to be over 3 years.

The final difference between the Berkeley and InterNiche versions of `t_select()` is the absence in the InterNiche version of the Berkeley `width` parameter. The `width` parameter is of use only when descriptor sets are represented as bit arrays and was thus deleted in the InterNiche implementation.

Returns

t_select() returns a non-negative value on success. A positive value indicates the number of ready descriptors in the descriptor sets. `0` indicates that the time limit specified by `tv` expired.

See Also: **t_accept(), t_connect(), t_listen(), t_recv(), t_send()**

Notes

Under rare circumstances, `t_select()` may indicate that a descriptor is ready for writing when in fact an attempt to write would block. This can happen if system resources necessary for a write are exhausted or otherwise unavailable. If an application deems it critical that writes to a file descriptor not block, it should set the descriptor for non-blocking I/O. See discussion of **t_setsockopt()**.

13.11 t_send, t_sendto

API Name

`t_send()`

`t_sendto()`

Syntax

```
int t_send(long s, char *buf, int len, int flags);
```

```
int t_sendto(long s, char *buf, int len, int flags, struct sockaddr *to,
int tolen);
```

Parameters

s	Socket identifier
buf	Start address of data to be sent
len	Length of data to be sent
flags	Flags for sending process (e.g., MSG_OOB)
to	Pointer to struct sockaddr_in structure containing addressing information for the remote end. IPv6 link local addresses on MULTI_HOMED systems should include a scopeID.
tolen	Length of sockaddr_in structure (bytes)

Description

`t_send()` and `t_sendto()` are used to transmit the message addressed by buf to another socket. `t_send()` may be used only when the socket is in a connected state, while `t_sendto()` may be used at any time, in which case the address of the target is given by the to parameter. The length of the message is given by `len`.

No indication of failure to deliver is implicit in a `t_send()`. Locally detected errors are indicated by a return value of `-1`.

If no messages space is available at the socket to hold the message to be transmitted, then `t_send()` normally blocks, unless the socket has been placed in non-blocking I/O mode. The `t_select()` call may be used to determine when it is possible to send more data.

The `flags` parameter may include one or more of the following:

```
#define MSG_OOB         0x1   /* process out-of-band data */
#define MSG_DONTROUTE   0x4   /* bypass routing, use direct interface */
```

The flag MSG_OOB is used to send "out-of-band" data on sockets that support this notion (e.g. SOCK_STREAM); the underlying protocol must also support "out-of-band" data. MSG_DONTROUTE is usually used only by diagnostic or routing programs.

Returns

The call returns the number of characters sent, or -1 if an error occurred. On failure, it sets an internal t_errno to one of the errors listed in Sockets Errors to indicate the error. The t_errno can be retrieved by a call to t_errno(s).

See Also

t_recv(), t_select(), t_getsockopt(), t_socket()

13.12 t_shutdown

API Name

`t_shutdown()`

Syntax

```
int t_shutdown(long s, int how);
```

Parameters

s	Socket identifier
how	Type of shutdown (SHUT_RD, SHUT_WR, or SHUT_RDWR)

Description

The `t_shutdown()` call causes all or part of a full-duplex connection on the socket associated with `s` to be shut down. If `how` is `0`, then further receives will be disallowed. If `how` is `1`, then further sends will be disallowed. If `how` is `2`, then further sends and receives will be disallowed.

Returns

This returns `0` on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_connect(), t_socket()

13.13 t_socket

API Name

`t_socket()`

Syntax

```
long t_socket (int domain, int type, int protocol);
```

Parameters

domain	Communication domain (AF_INET or AF_INET6)
type	Socket type (SOCK_STREAM or SOCK_DGRAM)
protocol	0

Description

`t_socket()` creates an endpoint for communication and returns a descriptor. The `domain` parameter specifies a communications domain within which communication will take place; this selects the `protocol` family which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file `socket.h`.

The socket has the indicated type, which specifies the semantics of communication. Currently defined types are:

```
SOCK_STREAM      /* TCP*/
SOCK_DGRAM       /* UDP */
SOCK_RAW         /* IP */
```

A `SOCK_STREAM` type provides sequenced, reliable, two-way connection based byte streams. A `SOCK_DGRAM` socket supports datagrams (connectionless, unreliable messages of a fixed, typically small, maximum length). A `SOCK_RAW` socket provides lower-layer protocol access.

Sockets of type `SOCK_STREAM` are full-duplex byte streams, similar to pipes. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a `t_connect()` call. Once connected, data may be transferred using `t_send()` and `t_recv()` calls. When a session has been completed, a `t_socketclose()` may be performed. Out-of-band data may also be transmitted as described in the `t_send()` page and received as described in `t_recv()`.

The communications protocols used to implement a SOCK_STREAM ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with -1 returns and with ETIMEDOUT as the specific code in the global variable t_errno. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (such as five minutes).

SOCK_DGRAM sockets allow sending of datagrams to correspondents named in t_sendto() calls. Datagrams are generally received with t_recvfrom(), which returns the next datagram with its return address.

SOCK_RAW sockets allow the application access to IP-layer protocols with a datagram-like interface; the application specifies the protocol of interest as the protocol argument to t_socket().

The operation of sockets is controlled by socket level options. These options are defined in the file socket.h. t_getsockopt() and t_setsockopt() are used to get and set options, respectively.

The SO_NOSLOWSTART option suppresses the standard TCP "slow-start" feature. Normally when newly connected, the TCP socket which is passed a large block of data to send (for example an FTP data connection) will send about two full-sized data segments, and then wait for a response from the other side before sending more. If the speed of the response indicates the network can handle more traffic, the connection will send more segments in reply. The number of segments will keep increasing until they are limited by internal resources or the receiver's window size.

In situations where a machine on an ethernet is sending it's packets through a router to a slower media (i.e. DSL) this slow start behaviour prevents the socket from flooding the router. The SO_NOSLOWSTART option defeats this feature, allowing the first data burst on the net to be the maximum number of segments allowed.

SO_FULLMSS prevents the socket from sending any data until the socket has buffered enough data to send a full sized packet. The size is determined by the network hardware, usually about 1460 data bytes. This should be used judiciously, since it may prevent proper operation of typical network applications. The problem is that an application command will not be sent until enough commands are buffered to produce the full-sized packet. For applications like FTP and HTTP, the average command is much too small to trigger the send. This option will not be available unless the TCP layer has been compiled with the #define SUPPORT_SO_FULLMSS in ipport.h.

The TCP_NODELAY disables the Nagle algorithm, and prevents attempts to coalesce small packets less than the TCP_MSS, while awaiting acknowledgement for data already sent.

TCP_ACKDELAYTIME sets the delay time for TCP delayed ACKs. The number of milliseconds of delay is passed to setsockopt() as a parameter. The millisecond time specified will by rounded off to the nearest "cticks" value. This option will not be available unless the TCP layer has been compiled with the #define DO_DELAY_ACKS in ipport.h. Builds which are compiled with this define will do delayed acks on all sockets by default, with a value of 1 ctick.

The TCP_NOACKDELAY option defeats delayed acking on specific sockets and will not be available unless the TCP layer has been compiled with the `#define DO_DELAY_ACKS` in `ipport.h`. Builds which are compiled with this define will do delayed acks on all sockets by default, with a value of 1 `ctick`. Setting TCP_NOACKDELAY will cause sockets to ack immediately as decided by the TCP code, with no delay.

TCP_MAXSEG is used to set the TCP MSS (Maximum Segment Size) value of the socket. Normally this value default to the size of the largest datagram supported on the underlying media, minus room for TCP, IP, and media headers. On ethernet this number is 1460 octets. This option can be called anytime after the socket is created, and should be called before the socket is connected. Calling after connection will produce unpredictable results. The value passed should generally be smaller than the default value as larger values may result in IP fragmentation.

Returns

`t_socket()` returns a non-negative descriptor on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_accept, t_bind, t_connect, t_getsockname, t_getsockopt, t_listen, t_recv, t_select, t_send, t_shutdown

13.14 t_socketclose

API Name

`t_socketclose()`

Syntax

```
int t_socketclose(long s);
```

Note: this is just `close()` on traditional Sockets systems.

Parameters

s	Socket identifier
name	Pointer to struct sockaddr_in structure containing addressing information for remote end (peer)
namelen	Length of sockaddr_in structure (bytes)

Description

The `t_socketclose()` call causes all of a full-duplex connection on the socket associated with `s` to be shut down and the socket descriptor associated with `s` to be returned to the free socket descriptor pool. Once a socket is closed, no further socket calls should be made with it.

Returns

This returns `0` on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_accept(), t_socket()

13.15 tcp_pktalloc

API Name

`tcp_pktalloc()`

Syntax

```
PACKET tcp_pktalloc(int datasize, int domain);
```

Parameters

```
int datasize /* size of TCP data for packet */

int domain /* AF_INET for IPv4, AF_INET6 for IPv6 */
```

Description

`tcp_pktalloc()` allocates a packet buffer large enough to hold datasize bytes of TCP data, plus TCP, IP, and MAC headers. It is a small wrapper around the internal `pk_alloc()` function that provides the necessary synchronization and calculation of header length.

`tcp_pktalloc()` should be called to allocate a buffer for sending data via `tcp_xout()`. It will return the allocated packet buffer with its `pkt->nb_prot` field set to where the application should deposit the data to be sent.

Returns

Returns a PACKET (pointer to `struct netbuf`) if OK, else `NULL` if a big enough packet was not available.

Notes

The `domain` field is ignored unless `ONEBUF` is defined, in which case `tcp_pktalloc()` will automatically add the length of the headers for the domain to the size of the packet buffer and `pkt->nb_prot` will be moved to the next byte beyond the headers to point to where the application should write its data.

See Also

tcp_pktfree(), tcp_xout()

13.16 tcp_pktfree

API Name

`tcp_pktfree()`

Syntax

```
void tcp_pktfree(PACKET p);
```

Description

`tcp_pktfree()` frees a packet allocated by (presumably) `tcp_pktalloc()` or passed to the application by a callback. This is a simple wrapper around `pk_free()` to lock and unlock the free-queue resource.

Parameters

```
PACKET p /* the pointer to the packet to be returned to the Protocol stack
*/
```

Returns

No value is returned. If the passed packet is already in a free queue, has been corrupted, or does not appear to be a valid packet, a `dtrap()` may be generated by the debugging logic.

See Also

tcp_pktalloc()

13.17 tcp_sleep, tcp_wakeup

API Name

`tcp_sleep()`

`tcp_wakeup()`

Syntax

```
void tcp_sleep(void *address);
```

```
void tcp_wakeup(void *address);
```

Description

These functions provide a mechanism by which the InterNiche TCP code can yield control of the target processor while waiting for one or more events to occur. The functions' address parameters provide a mechanism by which the source of the events can be synchronized.

See the detailed description of this in the TCP Sleep section of this document.

13.18 tcp_xout

API Name

`tcp_xout()`

Syntax

```
int tcp_xout(long s, PACKET pkt);
```

Parameters

```
long s /* socket on which packet is to be sent */

PACKET pkt /* pointer to packet to be sent */
```

Description

The `tcp_xout()` call sends a packet buffer on a socket. The packet buffer must be initialized with `pkt->nb_prot` pointing to the start of the application data to be sent (this will have been set properly by `tcp_pktalloc()`), and with `pkt->nb_plen` set to the number of bytes of data to be sent.

Returns

An integer indicating the success or failure of the function. A returned value of zero indicates that the packet was sent successfully. Returned values less than zero indicate errors, and that the packet was not accepted by the stack (so the application must either re-send the packet via a later call to `tcp_xout()` or free the packet via `tcp_pktfree()`). Returned values greater than zero indicate that the packet has been accepted and queued on the socket but has not yet been transmitted.

See Also

tcp_pktalloc(), tcp_pktfree()

14 SSL

14.1 sslclnt_app_init

```
API Name
sslclnt_app_init()
Syntax
SSLCLNT_APP_CTX *sslclnt_app_init(struct sslclnt_cfg *cfg);
Parameters
   cfg
Pointer to structure containing information about client configuration paramete



Description
This function creates a new SSL/TLS-based client application context.
Returns
This function returns a pointer to the newly created context.
```

14.2 sslclnt_app_term

API Name

`sslclnt_app_term()`

Syntax

```
int sslclnt_app_term(SSLCLNT_APP_CTX *actx);
```

Parameters

actx	Pointer to SSL/TLS-based client application's context structure (returned from a prior call to sslclnt_app_init())

Description

This function destroys a SSL/TLS-based client application context.

Returns

This function returns ESUCCESS if the cleanup was completed successfully; otherwise, it returns EFAILURE.

14.3 sslclnt_create_conn

API Name

`sslclnt_create_conn()`

Syntax

```
SSLCLNT_CONN *sslclnt_create_conn(SSLCLNT_APP_CTX *actx, SSLCLNT_CONN
*conn, SOCKTYPE sock, int *status);
```

Parameters

actx	Pointer to SSL/TLS-based client application's context
conn	Pointer to SSL/TLS-based client application's secure connection
sock	Identifier for TCP socket used for secure connection
status	Pointer to integer for storing the status of the process of establishment of the secure connection (output)

Description

This function creates a new SSL/TLS-based secure connection that can be used for I/O by the client. When using a non-blocking socket, 'conn' is NULL in the first invocation, and non-NULL (i.e., equal to the value returned from the first invocation) in all subsequent invocations. When using a blocking socket, 'conn' is set to NULL. The 'status' variable can be set (by this function) to any one of the following three values: SSLCLNT_CONNECT_INCOMPLETE, SSLCLNT_CONNECT_ERROR, or SSLCLNT_CONNECT_COMPLETE. When using non-blocking sockets, this function will set 'status' to SSLCLNT_CONNECT_INCOMPLETE until the SSL/TLS connection establishment process is complete. A status of SSLCLNT_CONNECT_ERROR indicates that the connection establishment process has failed. An application client can perform data I/O only after the SSL/TLS connection establishment process has successfully completed.

Returns

This function returns a pointer to the newly created connection.

14.4 sslclnt_del_conn

API Name

`sslclnt_del_conn()`

Syntax

```
int sslclnt_del_conn(SSLCLNT_CONN *conn);
```

Parameters

conn	Pointer to SSL/TLS-based client application's secure connection structure

Description

This function deletes all data structures associated with the specified SSL/TLS-based secure connection.

Returns

This function returns ESUCCESS if the deletion was successful; otherwise, it returns EFAILURE.

14.5 sslclnt_get_stats

API Name

`sslclnt_get_stats()`

Syntax

```
void sslclnt_get_stats(SSLCLNT_STATS *stats);
```

Parameters

stats	Pointer to SSL client statistics data structure

Description

This function copies the SSL client statistics into the caller-provided data structure. The statistics are collected across all SSL client-based applications.

Returns

None.

14.6 sslclnt_recv

API Name

`sslclnt_recv()`

Syntax

```
int sslclnt_recv(SSLCLNT_CONN *conn, char *buf, int length);
```

Parameters

conn	Pointer to SSL/TLS-based client application's secure connection
buf	Pointer to buffer for receiving data
length	Length of 'buf'

Description

This function is used to receive data on the secure connection.

Returns

This function returns the number of bytes received on the connection, 0 (e.g., if the connection has been closed), or -1 (EFAILURE).

14.7 sslclnt_send

API Name

`sslclnt_send()`

Syntax

```
int sslclnt_send(SSLCLNT_CONN *conn, char *buf, int length);
```

Parameters

conn	Pointer to SSL/TLS-based client application's secure connection
buf	Pointer to buffer containing data to be sent
length	Amount of data in 'buf'

Description

This function is used to transmit data on the secure connection.

Returns

This function returns the number of bytes written on the connection or -1 (EFAILURE).

14.8 sslclnt_term_conn

API Name

`sslclnt_term_conn()`

Syntax

```
int sslclnt_term_conn(SSLCLNT_CONN *conn, bool_t *bidirectional_shutdown);
```

Parameters

conn	Pointer to SSL/TLS-based client application's secure connection
bidirectional_shutdown	Pointer to boolean parameter (TRUE or FALSE) indicating if the shutdown was bidirectional(output)

Description

This function initiates the termination process for a SSL/TLS-based secure connection by sending a Close Notify alert to the peer.

Returns

This function returns ESUCCESS if the termination was successful; otherwise, it returns EFAILURE.

14.9 sslsrv_app_init

API Name

`sslsrv_app_init()`

Syntax

```
SSLSRV_APP_CTX *sslsrv_app_init(struct sslsrv_cfg *cfg, bool_t
create_socks);
```

Parameters

cfg	Pointer to structure containing information about server configuration parameters
create_socks	Boolean parameter indicating whether this function should create listening socket(s)

Description

This function creates a new SSL/TLS-based server application context (and listening sockets, if requested by caller).

Returns

This function returns a pointer to the newly created context.

14.10 sslsrv_app_term

API Name

`sslsrv_app_term()`

Syntax

```
int sslsrv_app_term(SSLSRV_APP_CTX *actx);
```

Parameters

actx	Pointer to SSL/TLS-based server application's context structure (returned from a prior call to sslsrv_app_init())

Description

This function destroys a SSL/TLS-based server application context.

Returns

This function returns ESUCCESS if the cleanup was completed successfully; otherwise, it returns EFAILURE.

14.11 sslsrv_create_conn

API Name

`sslsrv_create_conn()`

Syntax

```
SSLSRV_CONN *sslsrv_create_conn(SSLSRV_APP_CTX *actx, SSLSRV_CONN *conn,
SOCKTYPE sock, int *status);
```

Parameters

actx	Pointer to SSL/TLS-based server application's context
conn	Pointer to SSL/TLS-based server application's secure connection
sock	Identifier for TCP socket used for secure connection
status	Pointer to integer for storing the status of the process of establishment of the secure connection (output)

Description

This function creates a new SSL/TLS-based secure connection that can be used for I/O by the server. When using a non-blocking socket, 'conn' is NULL in the first invocation, and non-NULL (i.e., equal to the value returned from the first invocation) in all subsequent invocations. When using a blocking socket, 'conn' is set to NULL. The 'status' variable can be set (by this function) to any one of the following three values: SSL_ACCEPT_INCOMPLETE, SSL_ACCEPT_ERROR, or SSL_ACCEPT_COMPLETE. When using non-blocking sockets, this function will set 'status' to SSL_ACCEPT_INCOMPLETE until the SSL/TLS connection establishment process is complete. A status of SSL_ACCEPT_ERROR indicates that the connection establishment process has failed. An application server can perform data I/O only after the SSL/TLS connection establishment process has successfully completed.

Returns

This function returns a pointer to the newly created connection.

14.12 sslsrv_del_conn

API Name

`sslsrv_del_conn()`

Syntax

```
int sslsrv_del_conn(SSLSRV_CONN *conn);
```

Parameters

conn	Pointer to SSL/TLS-based server application's secure connection

Description

This function deletes all data structures associated with the specified SSL/TLS-based secure connection.

Returns

This function returns ESUCCESS if the deletion was successful; otherwise, it returns EFAILURE.

14.13 sslsrv_get_client_certs

API Name

`sslsrv_get_client_certs()`

Syntax

```
int sslsrv_get_client_certs(SSLSRV_CONN *conn, SSLSRV_X509CERT_CHAIN
**client_certs);
```

Parameters

conn	Pointer to SSL/TLS-based server application's secure connection
client_certs	Pointer to location to store a pointer to an array of SSLSRV_X509CERT_CHAIN structures (output)

Description

This function retrieves the X.509 certificates provided by the client. Each SSLSRV_X509CERT_CHAIN element contains the DER-formatted data and length of one X.509 certificate from the client's certificate chain.

Returns

This function returns the number of certificates in the client's certificate chain.

14.14 sslsrv_get_conn_err

API Name

`sslsrv_get_conn_err()`

Syntax

```
int sslsrv_get_conn_err(SSLSRV_CONN *conn);
```

Parameters

conn	Pointer to SSL/TLS-based server application's secure connection

Description

This function retrieves the error number (errno) associated with the TCP socket used for I/O by the secure connection.

Returns

This function returns the error number for the specified connection; if the connection is NULL, it returns EINVAL.

14.15 sslsrv_get_stats

API Name

`sslsrv_get_stats()`

Syntax

```
void sslsrv_get_stats(SSLSRV_STATS *stats);
```

Parameters

stats	Pointer to SSL server statistics data structure

Description

This function copies the SSL server statistics into the caller-provided data structure. The statistics are collected across all SSL server-based applications.

Returns

None.

14.16 sslsrv_recv

API Name

`sslsrv_recv()`

Syntax

```
int sslsrv_recv(SSLSRV_CONN *conn, char *buf, int length);
```

Parameters

conn	Pointer to SSL/TLS-based server application's secure connection
buf	Pointer to buffer for receiving data
length	Length of 'buf'

Description

This function is used to receive data on the secure connection.

Returns

This function returns the number of bytes received on the connection, 0 (e.g., if the connection has been closed), or -1 (EFAILURE).

14.17 sslsrv_send

API Name

`sslsrv_send()`

Syntax

```
int sslsrv_send(SSLSRV_CONN *conn, char *buf, int length);
```

Parameters

conn	Pointer to SSL/TLS-based server application's secure connection
buf	Pointer to buffer containing data to be sent
length	Amount of data in 'buf'

Description

This function is used to transmit data on the secure connection.

Returns

This function returns the number of bytes written on the connection or -1 (EFAILURE).

14.18 sslsrv_term_conn

API Name

`sslsrv_term_conn()`

Syntax

```
int sslsrv_term_conn(SSLSRV_CONN *conn, bool_t *bidirectional_shutdown);
```

Parameters

conn	Pointer to SSL/TLS-based server application's secure connection
bidirectional_shutdown	Pointer to boolean parameter (TRUE or FALSE) indicating if the shutdown was bidirectional(output)

Description

This function initiates the termination process for a SSL/TLS-based secure connection by sending a Close Notify alert to the peer.

Returns

This function returns ESUCCESS if the termination was successful; otherwise, it returns EFAILURE.

15 SYSLOG

15.1 openlogaddr, closelogfac

API Name

`openlogaddr()`

`closelogfac()`

Syntax

```
void openlogaddr(int facility,
                 char *iden,
                 int logopt,
                 struct sockaddr *sa,
                 int sa_len,
                 char *fname);

void closelogfac(int facility);
```

Parameters

facility	One of the facility codes defined in `h/syslog.h`
iden	Pointer to ID information
logopt	logging options (e.g., LOG_CONS, LOG_FILE)
sa	pointer to generic socket address structure
sa_len	length of socket address structure
fname	name of facility-specific logfile. Null if logopt = LOG_CONS

Description

The function `openlogadd()` is used to start logging to a particular syslog server. InterNiche syslog client allows separate logging for each facility. Hence different applications can use this feature to log to different syslog servers. The fname parameter gives the name of the file where messages are to be logged. When the application is done logging, it can call `closelogfac()` to close special logging for the particular facility.

Returns

Nothing

File

misclib/syslog.c

15.2 syslog, openlog, closelog, setlogmask

API Name

`syslog()`

`openlog()`

`closelog()`

`setlogmask()`

Syntax

```
void closelog (void);

void openlog (const char * ident, int logopt, int facility);

void syslog (int priority, const char * msg, ...);

int setlogmask (int maskpri);
```

Parameters

```
const char * ident; /* Identity of the application */

int logopt; /* Options for logging */

int facility; /* Application/facility doing the log */

int priority; /* Priority of the log */

int maskpri; /* Used to mask logs of lower priorities */
```

File

`misclib/syslog.c`

Description

The `syslog()` function writes message to the syslog server. The message is then written to the system console, log files, logged-in users, or forwarded to other machines as appropriate. The message is identical to a printf format string. ('%m' is supported by BSD, but not supported in this implementation). A trailing newline is added if none is present. The `vsyslog()` function of BSD is not supported. The message is tagged with priority. Priorities are encoded as a facility and a level. The facility describes the part of the system generating the message. The level is selected from the following ordered (high to low) list:

LOG_EMERG	A panic condition. This is normally broadcast to all users.
LOG_ALERT	A condition that should be corrected immediately, such as a corrupted system database.
LOG_CRIT	Critical conditions, e.g., hard device errors.
LOG_ERR	Errors.
LOG_WARNING	Warning messages.
LOG_NOTICE	Conditions that are not error conditions, but should possibly be handled specially.
LOG_INFO	Informational messages.
LOG_DEBUG	Messages that contain information normally of use only when debugging a program.

The `openlog()` function provides for more specialized processing of the messages sent by syslog(). The parameter ident is a string that will be prepended to every message. The logopt argument is a bit field specifying logging options, which is formed by OR'ing one or more of the following values:

LOG_CONS	If `syslog()` cannot pass the message to syslogd it will attempt to write the message to the console
LOG_NDELAY	Open the connection to `syslogd` immediately. Normally the open is delayed until the first message is logged. Useful for programs that need to manage the order in which file descriptors are allocated.
LOG_PERROR	Write the message to standard error output as well to the system log.
LOG_PID	Log the process id with each message: useful for identifying instantiations of daemons.

The facility parameter encodes a default facility to be assigned to all messages that do not have an explicit facility encoded:

LOG_AUTH	The authorization system
LOG_AUTHPRIV	The same as LOG_AUTH, but logged to a file readable only by selected individuals.
LOG_CONSOLE	Messages written to console by the kernel console output driver.
LOG_CRON	The cron daemon
LOG_DAEMON	System daemons that are not provided for explicitly by other facilities.
LOG_FTP	The file transfer protocol daemons
LOG_KERN	Messages generated by the kernel. These cannot be generated by any user processes.
LOG_LPR	The line printer spooling system
LOG_MAIL	The mail system.
LOG_NEWS	The network news system.
LOG_SECURITY	Security subsystems
LOG_SYSLOG	Messages generated internally by `syslogd()`
LOG_USER	Messages generated by random user processes. This is the default facility identifier if none is specified.
LOG_UUCP	The uucp system.
LOG_LOCAL0	Reserved for local use. Similarly for `LOG_LOCAL1` through `LOG_LOCAL7`.

The `closelog()` function can be used to close the log file.

The `setlogmask()` function sets the log priority mask to maskpri and returns the previous mask. Calls to syslog() with a priority not set in maskpri are rejected. The mask for an individual priority pri is calculated by the macro LOG_MASK(pri); the mask for all priorities up to and including toppri is given by the macro LOG_UPTO(toppri);. The default allows all priorities to be logged.

Returns

The routines `closelog()`, `openlog()`, `syslog()` return no value.

`setlogmask()` always returns the previous log mask level.

16 System

16.1 cksum

API Name

`cksum() - Calculate buffer checksum`

Syntax

```
unsigned short cksum (char *buffer, unsigned word_count);
```

Parameters

```
char *buffer /* pointer to buffer to checksum */

unsigned word_count /* number of 16 bit words in buffer */
```

Description

Returns 16 bit Internet checksum of buffer. Algorithms for this are described in RFC1071.

NOTE: A portable C language version of this routine is provided with the demo packages, however TCP implementations can spend a significant portion of their CPU cycles in the checksum routine. This routine is described here to encourage porting engineers to optimize their ports by implementing their checksum routines in assembly language. An Intel x86 assembly language checksum routine is also included which can be used on Intel processors as is. Versions for other CPUs are widely available - contact us if you need help finding one.

Returns

The 16 bit checksum.

16.2 create_device

API Name

`create_device()`

Syntax

```
int create_device(NET ifp, void * bindinfo);
```

Parameters

```
NET ifp /* interface descriptor (pointer to struct net) */

void * bindinfo /* driver-specific binding information for the device */
```

Description

A `create_device()` function must be supplied for drivers that are written to support dynamic network interfaces. This function must be passed to the stack's `ni_create()` function when creating a network interface; `ni_create()` will call this function so that the driver can bind or attach to a device and complete the initialization of the struct net for the device.

When the stack calls the driver's `create_device()` function, the ifp argument will be a pointer to the newly-created network interface's `struct net`, and the `bindinfo` argument will be the `bindinfo` argument that was passed to `ni_create()`. The `create_device()` function may use the bindinfo argument to locate any addressing or binding information that it needs to initialize the network interface device, and should perform initialization of the supplied struct net, as would need to be done by `prep_ifaces()` and `n_init()` for static network interfaces that are initialized at startup time.

Returns

Returns `0` if OK, else one of the `ENP_` codes.

16.3 eth_prep

API Name

`eth_prep() - Setup ethernet nets structure`

Syntax

```
int eth_prep(int index)
```

Parameters

`index`	interface number in the `nets` array

Description

Prepare the ethernet interface structure for this device, including populating the interface function pointers and flags for operation.

Returns

success(1) or failure(0)

16.4 get_pticks

API Name

`get_pticks() - Get fast timer tick count`

Syntax

```
uint32_t get_pticks(void)
```

Parameters

None.

Description

Counts time since power up. Frequency is PPS (default 100Hz)

Returns

Returns 32 bit tick count of time since power up.

16.5 ENTER_CRIT_SECTION, EXIT_CRIT_SECTION

API Name

`ENTER_CRIT_SECTION() - Enter critical section`

`EXIT_CRIT_SECTION() - Leave critical section`

Syntax

```
#define ENTER_CRIT_SECTION

#define EXIT_CRIT_SECTION
```

Parameters

None

Description

These two entry points should be designed to be paired around sections of code that must not be interrupted or pre-empted. Usually, `ENTER_CRIT_SECTION()` should save the processor interrupt state and disable interrupts, whereas `EXIT_CRIT_SECTION()` should restore the processor interrupt state to what it was before the most recent call to `ENTER_CRIT_SECTION()`.

See the detailed discussion of these macros see Critical Section Method.

Returns

These return no meaningful value.

Note

Nested calls to ENTER_CRIT_SECTION() and EXIT_CRIT_SECTION() **must** be supported.

16.6 LOCK_NET_RESOURCE, UNLOCK_NET_RESOURCE

API Name

`LOCK_NET_RESOURCE() - Resource access lock`

`UNLOCK_NET_RESOURCE() - Resource access unlock`

Syntax

```
void LOCK_NET_RESOURCE(int resID);

void UNLOCK_NET_RESOURCE(int resID);

void WAIT_NET_RESOURCE(int resID, int timeout);
```

Parameters

Any of the `xxx_RESID` constants.

Description

See description of Net Resource Method.

Returns

Nothing.

16.7 TK_BLOCK

API Name

`TK_BLOCK - Relinquish the CPU to another task`

Syntax

```
void TK_BLOCK();
```

Parameters

None.

Description

The macro is called by the current task when it has no more immediate work to do. In most systems, this macro is equivalent to TK_YIELD(). Execution of the current task is stopped and execution of the next task that is ready to run is started.

Notes/Status

See TK_YIELD() for further discussion.

Returns

Nothing

16.8 TK_CREATE

API Name

`TK_CREATE - Create a task`

Syntax

```
int TK_CREATE(void (*code)(void *), char *name, int stack, void *param, unsigne
```

Parameters

code	Pointer to the function to be called when the task is started. This function's prototype is: `void code(void *param)` In a tasking environment, this function should never return.
name	`NUL`-terminated string that is the name of the task.
stack	An integer specifying the size of the task's stack in bytes.
param	The parameter passed to the '`code`' function when the task is started.
prio	An unsigned integer specifying the task's execution priority. If the RTOS does not support task priorities, '`prio`' can be zero.
taskp	A pointer to a variable of type `TASK *`. If '`taskp`' is not NULL, the ID of the created task will be stored in '`taskp`'.

Description

`TK_CREATE()` is called to create a task. This may include allocating a task structure, a stack, or other resources for the task. Note that in some tasking systems the task structures and stack memory are statically declared and only need to be activated, while in others, such as NicheTask, the tasks and stacks are allocated from the heap. After the task is created, it is left in the `SUSPENDED` state; `TK_RESUME()` must be called to set the task to run.

Tasks may be started by the system at any time after creation. Tasks should be coded to test for any required resources or conditions as they start executing. An example of this is the netmain_mod.c application tasks, which test the global variable `iniche_net_ready` before commencing network I/O.

Returns

`TK_CREATE` returns `ESUCCESS` if the task was successfully created, and `EFAILURE` otherwise.

16.9 TK_DELETE

API Name

`TK_DELETE - Delete a task`

Syntax

```
void TK_DELETE(TASK tk);
```

Parameters

tk	Task ID of the task to delete. A value of `TK_THIS` is equivalent to the ID of the calling task.

Description

Terminates execution of the specified task and deletes the task's control structure and stack. If the task is self-destructing, execution will continue with the next task that is ready to run.

Returns

Nothing

16.10 TK_RESUME, TK_WAKE

API Name

`TK_RESUME - Resume execution of a suspended task`

`TK_WAKE - Resume execution of a suspended task`

Syntax

```
void TK_RESUME(TASK tk);
```

```
void TK_WAKE(TASK tk);
```

Parameters

tk	Task ID of the task to be woken up.

Description

This is called to awaken a task which has been suspended via `TK_SUSPEND()` or to start a task that has just been created. The task is marked ready to run. The task will not resume execution immediately unless it is of a higher priority than the current task.

Returns

Nothing

16.11 TK_SIGNAL

API Name

`TK_SIGNAL - Signal a task from another task`

Syntax

```
int TK_SIGNAL(IN_SEM s);
```

Parameters

s	s is a semaphore object.

Description

When a semaphore is signaled, all tasks that are waiting for the signal (see `TK_SIGWAIT()`) are set ready to be run. `TK_SIGNAL()` is similar to `TK_RESUME()` except that a signal can occur before a task is ready to wait for it. In that case, the signal is recorded, and the task's call to `TK_SIGWAIT()` will return immediately.

Interrupt handlers should use `TK_SIGNAL_ISR()` rather than `TK_SIGNAL()` to wake up a task due to the asynchronous timing between the calls to `TK_SIGNAL()` and `TK_SIGWAIT()`.

Returns

`ESUCCESS` if the signal was recorded successfully and `EFAILURE` if there is an error recording the signal.

16.12 TK_SIGNAL_ISR

API Name

`TK_SIGNAL_ISR - Signal a task from an interrupt routine`

Syntax

```
int TK_SIGNAL_ISR(IN_SEM s);
```

Parameters

s	Semaphore to be signaled.

Description

The semaphore is signaled. If a task is waiting for the semaphore via a call to `TK_SIGWAIT()`, the task will be set ready to run. If no task is waiting, then its next call to `TK_SIGWAIT()` will return immediately becasue the signal will already be present.

On preemptive systems, if the priority of the waiting task is higher than the priority of the interrupted task, a task switch may occur when the interrupt handler returns.

Returns

`TRUE` if a task switch will occur and `FALSE` if a task switch will not occur.

16.13 TK_SIGWAIT

API Name

`TK_SIGWAIT - Wait for a signal`

Syntax

```
int TK_SIGWAIT(IN_SEM s, long timeout);
```

Parameters

s	A semaphore object.
timeout	The number of cticks to wait for the signal to occur.

Description

Suspends the calling task until the specified semaphore is signaled or until the specified number of cticks has elapsed. If the semaphore has already been signaled, `TK_SIGWAIT()` returns immediately.

Returns

`ESUCCESS` if the signal was received before the timeout elapsed, `EFAILURE` if there was an error in signaling the semaphore and `TK_TIMEOUT` if the timeout elapsed before the signal was received.

16.14 TK_SLEEP

API Name

`TK_SLEEP - Pause a task for a period of time`

Syntax

```
void TK_SLEEP(unsigned long ticks);
```

Parameters

ticks	Number of CTICKs to wait before being scheduled.

Description

Execution of the calling task is suspended for the specified number of system clock ticks (`CTICKs`). On InterNiche networking systems, clock ticks are tracked by the variable cticks, and the frequency is defined by `TPS` (ticks per second).

Tasks put to sleep with this call may be awakened before the indicated time by a call to `TK_RESUME()`.

Returns

Nothing

16.15 TK_SUSPEND

API Name

`TK_SUSPEND - Suspend execution of a task`

Syntax

```
void TK_SUSPEND(TASK tk);
```

Parameters

tk	Task ID of the task to be suspended. A task ID value of TK_THIS refers to the current task.

Description

When a task is suspended, the task flags are set to not ready. If the task being suspended is the current task, it is as if the current task called `TK_BLOCK()`. The task will not be run again until another task or interrupt handler calls `TK_RESUME()` with the suspended task's ID.

Returns

Nothing

16.16 TK_YIELD, tk_yield

API Name

`TK_YIELD () - Relinquish the CPU to another tasktk_yield() - Relinquish the CPU to another task`

Syntax

```
void TK_YIELD(void);
```

Parameters

None

Description

`TK_YIELD()` is called when the task code wants to wait for something to occur - a situation often referred to as a "busy wait". The `TK_YIELD()` primitive must give other tasks a chance to run, yet resume the calling task in a short interval. On a round-robin system like NicheTask this is easy - you simply mark to current task as runnable an call the round-robin scheduled.

On an RTOS where tasks have priorities, this can be somewhat trickier to implement. These systems sometimes support a call which will let tasks of equal or greater priority run, by not lower priority tasks. A task spinning on such a `TK_YIELD()` macro would never allow a lower priority task to run.

One remedy for this is to code the `TK_YIELD()` macro to put the task to sleep for a single clock tick. This will force it to wait a reasonable interval during which lower priority tasks may potentially get some cycles. The draw back is that even when the system has nothing else to do, the task spinning on will never be able to utilize all the CPUs power - it will always spend a certain amount of time gratuitously blocked.

Notes

The `tk_yield()` macro (same name in lower case) is identical to the uppercase version. It is supported for historical reasons.

Returns

Nothing

16.17 npalloc, npfree

API Name

`npalloc`

`npfree`

Syntax

```
void *npalloc(int size);

void *npfree(void *);
```

Description

All the IP stack's dynamic memory is allocated by calls to `npalloc()` and released by calls to `npfree()`. The syntax for these is exactly the same as the standard C library calls `malloc()` and `free()`, with the exception that buffers returned from `npalloc()` are assumed to be pre-initialized to all zeros. In this respect `npalloc()` is like `calloc()`.

If the target system already supports standard `calloc()` and `free()` calls, all that is necessary is to add the following lines to `ipport.h`:

```
#define npalloc(size) calloc(1, size) #define npfree(ptr) free(ptr)
```

In the event your target system does not support `calloc()` and `free()`, you will need to implement them. An exhaustive description of how these functions work and sample code is available in "The C Programming Language" by Kernighan and Ritchie.

The great majority of the calls to `npalloc()` are made at initialization time. Only the UDP and TCP layers require these calls during runtime. If your system has severe memory shortages, then these layers can be modified to use pre-allocated blocks of static memory rather than implement a fully functional `npalloc()` and `npfree()`, but this is invariably more work and puts limits on the number of simultaneous connections which can be supported.

One issue that must be dealt with on some target processors is memory alignment. Some processors will generate faults if instructions to read or write more than one byte of memory at a time from odd numbered addresses are executed. Even if the target processor supports 2 or 4 byte reads and writes to odd numbered addresses, instructions of this sort usually execute more slowly than accesses to addresses that are an integer multiple of the number of bytes accessed by the instruction. System performance can suffer. If the target system supports the standard `calloc()` and `free()` calls then the C library vendors have probably already made sure that the buffers returned by `calloc()` are properly aligned for the target processor. However if you need to implement `npalloc()` and `npfree()` without the standard C library memory allocation functions then you should implement these functions such that the memory blocks are aligned on addresses that are a multiple of the data bus width of the target processor. If you don't know the data bus width of the target processor, 4 is usually a safe guess.

Returns

`npalloc()` returns a pointer to the block allocated or `NULL` if memory is unavailable.

16.18 sysuptime

API Name

`sysuptime()`

Syntax

```
unsigned long sysuptime(void);
```

Parameters

None

Description

Returns the number of 1/100ths of a second that have elapsed since the target system was last booted.

Returns

See above.

Example

```
unsigned long sysuptime()
{
   return ((cticks/TPS)*100);   /* 100ths of a sec since boot time */
}
```

17 VFS

17.1 vclearerr

API Name

()

Syntax

```
void vclearerr(VFILE *vfd);
```

Description

vclearerr() clears the error condition returned by vferror().

Returns

Nothing.

17.2 vfclose

API Name

`vfclose()`

Syntax

```
void vfclose(VFILE * vfd);
```

Description

Files that are opened with `vfopen()` should eventually be closed with vfclose(). Depending on how the VFS has been configured and whether any changes to the file have been made since it was opened, a call to vfclose() can cause the function `vfs_sync()` to be called which allows for the RAM resident VFS to be stored to the target system's backing store.

Returns

Nothing

17.3 vferror

API Name

`vferror()`

Syntax

```
int vferror(VFILE * vfd);
```

Returns

vferror() returns an error code describing what went wrong on the last attempt to write to the file.

17.4 vfopen

API Name

`vfopen()`

Syntax

```
VFILE* vfopen(char * name, char * mode);
```

Description

The calling semantics of `vfopen()` are similar to that of the standard C library `fopen()`. The name parameter points to a null terminated string that defines the name of the file to be opened. The first character of the string addressed by the mode parameter defines what actions are to be taken when opening the file, as shown below:

`mode [0] == 'r'`	If the named file does not exist, fail the open. If the file does exist, open the file and position the `CFP` to the beginning of the file.
`mode [0] == 'w'`	If the named file does not exist, create a file of 0 length with the given name and open it. If the named file does exist, truncate it to a length of 0 and open it. In both cases position the `CFP` to the beginning of the file.
`mode [0] == 'a'`	If the named file does not exist, create a file of 0 length with the given name and open it. If the named file does exist, open it without modifying its existing contents. In both cases position the `CFP` to the end of the file.

Returns

When `vfopen()` is successful, it returns a handle which is a pointer to the type `VFILE`. This handle should be passed to subsequent VFS functions which require a `VFILE` parameter to access the file's contents. When `vfopen()` is not successful it returns `NULL` and the reason for the error can be retrieved by calling the function `get_vfopen_error()`..

Notes

`vfopen()` differs from the Standard `fopen()` call in the following ways:

- Only the first character of the mode parameter is significant. The 'b' and '+' suffixes that have special meaning in some `fopen()` implementations have no meaning to `vfopen()`. This means that the "open for read access only" semantic of the 'r' parameter that is present in `fopen()` does not apply. Writes to a file that is `vfopen()`'ed with mode 'r' will not automatically fail like they do on some systems. In that sense 'r' with `vfopen()` is more like 'r+' on most system's `fopen()`. It also means that the 'ASCII' mode of file opening in which newline conversion is done in the API is not performed with the VFS. All reads and writes are strictly binary.
- The VFS supports only one current file pointer per `VFILE`. Some buffered I/O systems will do reads from the "current file pointer" which is settable with `fseek()` but will only allow writes to the end of the file (as weird a "standard" behavior as one can imagine). With the VFS, reads and writes are always initiated from the `CFP`.
- The VFS imposes no requirements on file names other than that they are not to exceed `FILENAMEMAX` characters in length. Embedded spaces and punctuation characters are legal, as are ASCII characters with the most significant bit set. A file name of 0 length is legal. Slash (forward slash), '/', and backslash, '\', have no special meaning. The one exception to this is that if a file name begins with a slash, '/', it will be removed from the file name before the file is created. Thus the file names /foo and foo refer to the same file.

17.5 vfread

API Name

`vfread()`

Syntax

```
int vfread(char * buf, unsigned size, unsigned items, VFILE * vfd);
```

Description

The calling semantics of `vfread()` are similar to that of the standard `fread()`. An attempt to read the product of items times size bytes from the `CFP` of the `VFILE` addressed by the vfd parameter into the caller supplied buffer addressed by the buf parameter is made. If at least that many bytes are available in the file starting at the `CFP`, the call succeeds and returns items to the caller. If less than that many bytes are available, as much as is available is copied to the caller's buffer and the number of bytes copied divided by size is returned to the caller. This is an integer division, which implies that if it is important to know how many bytes were actually read, size should be 1. In all cases the `CFP` is incremented by the number of bytes successfully read.

Returns

The number of items successfully read into the caller's buffer.

17.6 vfseek

API Name

`vfseek()`

Syntax

```
long vfseek(VFILE * vfd, long offset, int mode);
```

Description

The calling syntax of vfseek() is similar to that of the standard C library fseek(), however the semantics are quite restricted. vfseek() allows the caller to change the `CFP` of a `VFILE`. The offset parameter must be 0. Two values are accepted for the mode parameter: SEEK_SET and SEEK_END. Thus vfseek() allows the caller to position the `CFP` to either the beginning (SEEK_SET) or the end (SEEK_END) of the file.

Returns

vfseek() returns the value of the modified `CFP` when successful. It returns -1 when unsuccessful. The reasons for failure usually have to do with invalid parameter values.

17.7 vftell

API Name

`vftell()`

Syntax

```
long vftell(VFILE * vfd);
```

Returns

For uncompressed files, `vftell()` functions much as the standard `ftell()`. It returns the `CFP` of the specified `VFILE`. For compressed files, `vftell()` returns the uncompressed size of the file if the `CFP` is at the end of the file, else is returns the byte offset into the compressed file image of the current `CFP`. File compression is described in the section, "Internal Data Structures".

17.8 vfwrite

API Name

`vfwrite()`

Syntax

```
int vfwrite(char * buf, unsigned size, unsigned items, VFILE * vfd);
```

Description

The calling semantics of `vfwrite()` are similar to that of the standard fwrite(). An attempt to write the product of items times size bytes from the caller's buffer addressed by the buf parameter to the `CFP` of the `VFILE` addressed by the vfd parameter is made. When successful, the `CFP` is incremented by the number of bytes written.

Returns

Because of its implementation, calls to `vfwrite()` either succeed completely and return items, or fail completely and return 0 to indicate that the file's contents were not modified. There is a possible exception to this when an external or local file system is used. The reason for the failure can be determined via a call to the vferror() function. The set of errors includes:

ENP_LOGIC	An attempt was made to do a write to a VFS in which write access is not enabled (`HT_RWVFS` is not defined).
ENP_FILEIO	An attempt was made to do a write to a VFS file that is write protected. Write protection of individual files is described later.
ENP_NOMEM	There was insufficient memory available to store the added file contents.

17.9 vgetc

API Name

`vgetc()`

Syntax

```
int vgetc(VFILE * vfd);
```

Returns

vgetc() returns the value of the byte at the current CFP and increments the CFP. It returns EOF (-1) when the end of the file is reached.

17.10 vunlink

API Name

`vunlink()`

Syntax

```
int vunlink(char *name);
```

Description

`vunlink()` deletes the named file from the set of files maintained by the VFS. Depending on how the VFS has been configured, a call to vunlink() can cause the function `vfs_sync()` to be called which allows for the RAM resident VFS to be stored to the target system's backing store.

Returns

`0` if the file was successfully deleted, `-1` otherwise.

The reasons for failure are:

- The named file does not exist in the VFS.
- The named file exists but was not marked as writable.

`vunlink()` modifies the parameter in the same manner as does `vfopen()`.