The Industria Libraries Manual

Göran Weinholt

This manual is for the Industria libraries, a collection of R6RS Scheme libraries.

Table of Contents

1 Getting started

1.1 Installation

The simple version: install Akku from `https://akkuscm.org/`. Run `akku install` in the your project directory. This fetches the Industria and its dependencies and install them to `.akku`. Run `.akku/env` to start using Industria in your R6RS Scheme of choice.

The longer, more manual, version: extend your Scheme library search path to include the `industria` directory, e.g. if you're using Chez Scheme on a Unix system and you unpacked Industria in `~/scheme`:

```
export CHEZSCHEMELIBDIRS=$HOME/scheme
```

Other possible environment variables include `IKARUS_LIBRARY_PATH`, `LARCENY_LIBPATH`, `MOSH_LOADPATH` and `YPSILON_SITELIB`. For more details please refer to your Scheme implementation's documentation. An alternative is to move or symlink the `industria` directory into a directory that already exists in your Scheme's search path. Download the dependencies noted in the Akku.lock file and install them in the library path.

Releases and the latest source code for Industria are available at GitHub `https://github.com/weinholt/industria/`.

1.2 Usage

To load an R6RS library into your program or library, put it in the `import` specification. Here's Hello World for R6RS Scheme:

```
#!/usr/bin/env scheme-script
(import (rnrs))
(display "Hello World!\n")
```

The first line is useful on Unix systems, but it is specified in the R6RS Non-Normative Appendices, so your Scheme might not accept programs with that line present.

See the `programs/` and `tests/` directories for examples.

2 Library reference

2.1 Cryptographic primitives

Beware that if you're using some of these libraries for sensitive data, let's say passwords, then there is probably no way to make sure a password is ever gone from memory. There is no guarantee that the passwords will not be swapped out to disk or transmitted by radio. There might be other problems as well. The algorithms themselves might be weak. Don't pick weak keys. Know what you're doing.

Your Scheme's implementation of `(srfi :27 random-bits)` might be too weak. It's common that it will be initialized from time alone, so an attacker can easily guess your `random-source` internal state by trying a few timestamps and checking which one generates the data you sent. These libraries try to use `/dev/urandom` if it exists, but if it doesn't they fall back on SRFI-27 and could reveal the secret of your heart to the enemy. See RFC4086 for details on how randomness works.

Please remember that these are low-level primitives. To be useful they must be part of a protocol. And remember what the license says about warranties.

2.1.1 Advanced Encryption Standard

The `(industria crypto aes)` library provides an implementation of the symmetrical Rijndael cipher as parameterized by the Advanced Encryption Standard (AES). It was created by the Belgian cryptographers Joan Daemen and Vincent Rijmen. Key lengths of 128, 192 and 256 bits are supported.

The code uses clever lookup tables and is probably as fast as any R6RS implementation of AES can be without using an FFI. The number of modes provided is pretty sparse though (only ECB and CTR). It also leaks key material via memory.

AES modes have recommendations for how many times they should be used with the same keying data.

`expand-aes-key` *key* [Procedure]

Expands the *key* into an *AES key schedule* suitable for `aes-encrypt!`. The *key* must be a bytevector of length 16, 24 or 32 bytes. The type of the return value is unspecified.

`aes-encrypt!` *source source-start target target-start key-schedule* [Procedure]

Takes the 16 bytes at *source*+*source-start*, encrypts them in Electronic Code Book (ECB) mode using the given *key-schedule*, and then writes the result at *target*+*target-start*. The *source* and the *target* can be the same.

```
(import (industria crypto aes))
(let ((buf (string->utf8 "A Scheme at work"))
      (sched (expand-aes-key (string->utf8 "super-secret-key"))))
  (aes-encrypt! buf 0 buf 0 sched)
  buf)
⇒ #vu8(116 7 242 187 114 235 130 138 166 39 24 204 117 224 5 8)
```

It is generally not a good idea to use ECB mode alone.

`reverse-aes-schedule` *key-schedule* [Procedure]

Reverses the *key-schedule* to make it suitable for `aes-decrypt!`.

`aes-decrypt!` *source source-start target target-start key-schedule* [Procedure]

Performs the inverse of `aes-encrypt!`. The *key-schedule* should first be reversed with `reverse-aes-schedule`.

```
(import (industria crypto aes))
(let ((buf (bytevector-copy #vu8(116 7 242 187 114 235 130 138
                                 166 39 24 204 117 224 5 8)))
      (sched (reverse-aes-schedule
              (expand-aes-key
               (string->utf8 "super-secret-key")))))
  (aes-decrypt! buf 0 buf 0 sched)
  (utf8->string buf))
⇒ "A Scheme at work"
```

`clear-aes-schedule!` *key-schedule* [Procedure]

Clears the AES key schedule so that it no longer contains cryptographic material. Please note that there is no guarantee that the key material will actually be gone from memory. It might remain in temporary numbers or other values.

`aes-ctr!` *source source-start target target-start len key-schedule ctr* [Procedure]

Encrypts or decrypts the *len* bytes at *source*+*source-start* using Counter (CTR) mode and writes the result to *target*+*target-start*. The *len* does not need to be a block multiple. The *ctr* argument is a non-negative integer.

This procedure is its own inverse and the *key-schedule* should not be reversed for decryption.

Never encrypt more than once using the same *key-schedule* and *ctr* value. If you're not sure why that is a bad idea, you should read up on CTR mode.

`aes-cbc-encrypt!` *source source-start target target-start k key-schedule iv* [Procedure]

Encrypts *k* bytes in the bytevector *source* starting at *source-start* with AES in CBC mode and writes the result to *target* at *target-start*.

The argument *k* must be an integer multiple of 16, which is the block length.

The *iv* bytevector is an Initial Vector. It should be 16 bytes long, initialized to random bytes. This procedure updates the *iv* after processing a block.

`aes-cbc-decrypt!` *source source-start target target-start k key-schedule iv* [Procedure]

The inverse of `aes-cbc-encrypt!`.

`make-aes-gcm-state` *key-schedule iv-length tag-length* [Procedure]

Creates an AES GCM state object that is used by the procedures below. The *key-schedule* argument is the key schedule created by `expand-aes-key`.

`aes-gcm-encrypt!` *source source-start target target-start k gcm-state iv aad tag tag-start* [Procedure]

Encrypts *k* bytes in the bytevector *source* starting at *source-start* with AES in Galois/Counter Mode (GCM) mode and writes the result to *target* at *target-start*. Source and target may overlap.

The argument *k* must be an exact integer. There is no block length for this mode.

The *gcm-state* argument must be an object created by `make-aes-gcm-state`.

The *iv* bytevector is an initial vector. It must never be reused with the same key.

GCM is different from CBC and CTR in that it offers Authenticated Encryption with Associated Data (*AEAD*). This means that it does not need to be combined with a separate HMAC procedure. The *aad* argument is a bytevector that contains additional data to be authenticated (or signed). The authentication tag (or signature) is written to the bytevector *tag* starting at length *tag-start*.

```
(import (rnrs)
        (industria crypto aes))

(let* ((key #vu8(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16))
       (iv-length 12)
       (tag-length 16)
       (gcm-state (make-aes-gcm-state (expand-aes-key key)
                                      iv-length tag-length)))
  (let* ((plaintext
          (string->utf8 "This is our message to you"))
         (ciphertext
          (make-bytevector (bytevector-length plaintext)))
         (iv (make-bytevector iv-length 0))
         (aad (string->utf8 "aad"))
         (tag (make-bytevector tag-length)))
    (bytevector-u32-set! iv 0 42 (endianness big))
    (aes-gcm-encrypt! plaintext 0 ciphertext 0
                      (bytevector-length plaintext)
                      gcm-state iv aad tag 0)
    (display ciphertext)
    (newline)
    (let ((decrypted
           (make-bytevector (bytevector-length plaintext))))
      ;; The assertion will fail if ciphertext, aad and tag
      ;; do not match up.
      (assert (eq? 'ok (aes-gcm-decrypt!?
                        ciphertext 0 decrypted 0
                        (bytevector-length ciphertext)
                        gcm-state iv aad tag 0)))
      (utf8->string decrypted))))
⊣ #vu8(202 108 237 127 3 64 111 82 76 90 193 152 36
       47 45 169 38 116 185 13 217 240 2 127 246 15)
⇒ "This is our message to you"
```

`aes-gcm-decrypt!?` *source source-start target target-start k gcm-state iv aad tag tag-start* [Procedure]

Encrypts *k* bytes in the bytevector *source* starting at *source-start* with AES in Galois/Counter Mode (GCM) mode and writes the result to *target* at *target-start*.

The other arguments are as for `aes-gcm-encrypt!`.

Note: the return value must be checked by the caller. If the symbol `ok` is returned then the data was authenticated correctly.

2.1.2 ARCFOUR stream cipher

The `(industria crypto arcfour)` library provides the well-known ARCFOUR stream cipher. It is the fastest of the ciphers provided by this library collection.

Since this is a stream cipher there is no block length.

`expand-arcfour-key` *key* [Procedure]

Expands the bytevector *key* into an ARCFOUR keystream value. The return value has an unspecified type and is suitable for use with the other procedures exported by this library.

Never use the same key to encrypt two different plaintexts.

`arcfour!` *source source-start target target-start k keystream* [Procedure]

Reads *k* bytes from *source* starting at *source-start*, XORs them with bytes from the *keystream*, and writes them to *target* starting at *target-start*. If *source* and *target* are the same object then it is required that *target-start* be less then or equal to *source-start*.

```
(import (industria crypto arcfour))
(let ((buf #vu8(90 60 247 233 181 200 38 52 121 82 133
                98 244 159 12 97 90 157 43 183 249 170
                73 244 126))
      (keystream (expand-arcfour-key
                  (string->utf8 "hardly a secret"))))
  (arcfour-discard! keystream 3000)
  (arcfour! buf 0 buf 0 (bytevector-length buf) keystream)
  (clear-arcfour-keystream! keystream)
  (utf8->string buf))
⇒ "I AM POKEY THE PENGUIN!!!"
```

`arcfour-discard!` *keystream n* [Procedure]

Discards *n* bytes from the keystream *keystream*. It is recommended that the beginning of the keystream is discarded. Some protocols, e.g. RFC 4345, require it.

`clear-arcfour-keystream!` *keystream* [Procedure]

Removes all key material from the *keystream*.

2.1.3 The Blowfish Cipher

The `(industria crypto blowfish)` library is a complete implementation of Bruce Schneier's Blowfish cipher. It is a symmetric block cipher with key length between 8 and 448 bits. The key length does not affect the performance.

`expand-blowfish-key` *key* [Procedure]
Expands a Blowfish *key*, which is a bytevector of length between 1 and 56 bytes (the longer the better). The returned key schedule can be used with `blowfish-encrypt!` or `reverse-blowfish-schedule`.

`blowfish-encrypt!` *source source-index target target-index schedule* [Procedure]
Encrypts the eight bytes at *source*+*source-start* using Electronic Code Book (ECB) mode. The result is written to *target*+*target-start*.

`reverse-blowfish-schedule` [Procedure]
Reverses a Blowfish key schedule so that it can be used with `blowfish-decrypt!`.

`blowfish-decrypt!` *source source-index target target-index schedule* [Procedure]
The inverse of `blowfish-encrypt!`.

`clear-blowfish-schedule!` [Procedure]
Clears the Blowfish key schedule so that it no longer contains cryptographic material. Please note that there is no guarantee that the key material will actually be gone from memory. It might remain in temporary numbers or other values.

`blowfish-cbc-encrypt!` *source source-start target target-start k schedule iv* [Procedure]
Encrypts *k* bytes in the bytevector *source* starting at *source-start* with Blowfish in CBC mode and writes the result to *target* at *target-start*.

The argument *k* must be an integer multiple of 8, which is the block length.

The *iv* bytevector is an Initial Vector. It should be 8 bytes long, initialized to random bytes. This procedure updates the *iv* after processing a block.

`blowfish-cbc-decrypt!` *source source-start target target-start k schedule iv* [Procedure]
The inverse of `blowfish-cbc-encrypt!`.

2.1.4 ChaCha20 stream cipher

The `(industria crypto chacha20)` library provides the ChaCha20 stream cipher, a 20-round variant of djb's ChaCha20 specified by RFC 7539.

Since this is a stream cipher there is no block length.

`chacha20-block!` *out key block-count nonce* [Procedure]
Update the first 64 bytes of the bytevector *out* with the output from the ChaCha20 block function. The *key* is a 256-bit key given as a 16-byte bytevector. The *block-count* is a 32-bit exact integer. The *nonce* is a 96-bit *number used only once* (sometimes called an IV), given as a 48-byte bytevector.

`chacha20-keystream` *key block-count nonce* [Procedure]
Opens a binary input port that yields the ChaCha20 keystream. The arguments are the same as for `chacha20-block!`.

`chacha20-encrypt!` *source source-start target target-start len keystream* [Procedure]

Copies *len* bytes from *source* to *target*, having the same interface as `bytevector-copy!`, except that it encrypts the bytes using the output from *keystream*.

The encryption is symmetric, so this is also the decryption primitive.

`chacha20-encrypt` *source source-start len keystream* [Procedure]

This is a convenience procedure that allocates a bytevector, but otherwise performs the same operation as `chacha20-encrypt`.

2.1.5 Elliptic Curve Diffie-Hellman key exchange

The `(industria crypto ecdh)` library exports procedures for Elliptic Curve Diffie-Hellman key exchange. The library is based on the X25519 and X448 functions from RFC 7748. ECDH is similar to DH, see Section 2.1.8 [crypto dh], page 10, but uses Curve25519 or Curve448. It is used when two parties want to generate a key without sending the actual key over the network.

The same caveats that apply to regular DH also apply to ECDH.

`make-ecdh-curve25519-secret` [Procedure]

Generates a Curve25519 private key *a* and a public key *K_A*.

`make-ecdh-curve448-secret` [Procedure]

Generates a Curve448 private key *a* and a public key *K_A*.

`ecdh-curve25519` *a K_B* [Procedure]

Compute the shared key given Alice's private key *a* and Bob's public key *K_B*.

```
(import (industria crypto ecdh))

(let-values ([(a K_A) (make-ecdh-curve25519-secret)]
             [(b K_B) (make-ecdh-curve25519-secret)])
  ;; The bytevectors being compared are the shared secret
  (equal? (ecdh-curve25519 a K_B)
          (ecdh-curve25519 b K_A)))
⇒ #t
```

`ecdh-curve448` *a K_B* [Procedure]

Compute the shared key given Alice's private key *a* and Bob's public key *K_B*.

`X25519` *k u* [Procedure]

The X25519 function, which is normally not needed, but is exposed for those who want to use it. It is a scalar multiplication on the Montgomery form of Curve25519.

`X448` *k u* [Procedure]

Same as `X25519`, but with Curve448.

2.1.6 Edwards-curve Digital Signature Algorithm (EdDSA)

EdDSA is a public key signature algorithm based on the edwards25519 and edwards448 curves. It is used in place for e.g. RSA signatures or DSA. Currently only the Ed25519 variant is supported.

`make-ed25519-public-key` *bv* [Procedure]
Make an Ed25519 public key using *bv*, which is 32 bytes long.

`ed25519-public-key?` *obj* [Procedure]
True if *obj* is an Ed25519 public key.

`ed25519-public-key=?` *key1 key2* [Procedure]
True if *key1* is the same Ed25519 public key as *key2*.

`ed25519-public-key-value` *key* [Procedure]
The public key bytevector for *key*.

`make-ed25519-private-key` *bv* [Procedure]
Make an Ed25519 private key using the secret *bv*, which is 32 bytes long.

`ed25519-private-key?` *obj* [Procedure]
True if *obj* is an Ed25519 private key.

`ed25519-private-key-secret` [Procedure]
The private key bytevector for *key*.

`ed25519-private->public` *key* [Procedure]
Returns the public key that corresponds to the Ed25519 private *key*.

`eddsa-private-key-from-bytevector` *bv* [Procedure]
Decodes *bv* as a private EdDSA key. This is intended to also support Ed448 in the future.

`ed25519-sign` *private-key msg* [Procedure]
Generate a SHA-512 based Ed25519 signature of *msg* using *private-key*.

`ed25519-verify` *public-key msg signature* [Procedure]
Verify a SHA-512 based Ed25519 *signature* of *msg* that was purportedly made with *public-key*. Returns `#t` iff the signature is valid.

2.1.7 Data Encryption Standard

The Data Encryption Standard (DES) is older than AES and uses shorter keys. To get longer keys the Triple Data Encryption Algorithm (TDEA, 3DES) is commonly used instead of DES alone.

The `(industria crypto des)` library is incredibly inefficient and the API is, for no good reason, different from the AES library. You should probably use AES instead, if possible.

`des-key-bad-parity?` *key* [Procedure]
Returns `#f` if the DES *key* has good parity, or the index of the first bad byte. Each byte of the *key* has one parity bit, so even though it is a bytevector of length eight (64 bits), only 56 bits are used for encryption and decryption. Parity is usually ignored.

`des!` *bv key-schedule* [*offset E*] [Procedure]
The fundamental DES procedure, which performs both encryption and decryption in Electronic Code Book (ECB) mode. The eight bytes starting at *offset* in the bytevector *bv* are modified in-place.

The *offset* can be omitted, in which case 0 is used.

The *E* argument will normally be omitted. It is only used by the `des-crypt` procedure.

```
(import (industria crypto des))
(let ((buf (string->utf8 "security"))
      (sched (permute-key (string->utf8 "terrible"))))
  (des! buf sched)
  buf)
⇒ #vu8(106 72 113 111 248 178 225 208)

(import (industria crypto des))
(let ((buf (bytevector-copy #vu8(106 72 113 111 248 178 225 208)))
      (sched (reverse (permute-key (string->utf8 "terrible")))))
  (des! buf sched)
  (utf8->string buf))
⇒ "security"
```

`permute-key` *key* [Procedure]
Permutes the DES *key* into a key schedule. The key schedule is then used as an argument to `des!`. To decrypt, simply reverse the key schedule. The return value is a list.

`tdea-permute-key` *key1* [*key2 key3*] [Procedure]
Permutes a 3DES key into a key schedule. If only one argument is given then it must be a bytevector of length 24. If three arguments are given they must all be bytevectors of length eight.

The return value's type is unspecified.

`tdea-encipher!` *bv offset key* [Procedure]
Encrypts the eight bytes at *offset* of *bv* using the given 3DES key schedule.

`tdea-decipher!` *bv offset key* [Procedure]
The inverse of `tdea-encipher!`.

`tdea-cbc-encipher!` *bv key iv offset count* [Procedure]
Encrypts the *count* bytes at *offset* of *bv* using Cipher Block Chaining (CBC) mode.

The *iv* argument is the *Initial Vector*, which is XOR'd with the data before encryption. It is a bytevector of length eight and it is modified for each block.

Both *offset* and *count* must be a multiples of eight.

`tdea-cbc-decipher!` *bv key iv offset count* [Procedure]
The inverse of `tdea-cbc-encipher!`.

`des-crypt` *password salt* [Procedure]

This is a password hashing algorithm that used to be very popular on Unix systems, but is today too fast (which means brute forcing passwords from hashes is fast). The *password* string is at most eight characters.

The algorithm is based on 25 rounds of a slightly modified DES.

The *salt* must be a string of two characters from the alphabet `#\A`–`#\Z`, `#\a`–`#\z`, `#\0`–`#\9`, `#\.` and `#\/`.

```
(import (industria crypto des))
(des-crypt "password" "4t")
⇒ "4tQSEW3lEnOio"
```

A more general interface is also available, see Section 2.5.4 [password], page 51.

2.1.8 Diffie-Hellman key exchange

The `(industria crypto dh)` library exports procedures and constants for Diffie-Hellman (Merkle) key exchange. D-H works by generating a pair of numbers, sending one of them to the other party, and using the other one and the one you receive to compute a shared secret. The idea is that it's difficult for an eavesdropper to deduce the shared secret.

The D-H exchange must be protected by e.g. public key encryption because otherwise a MITM attack is trivial. It is best to use a security protocol designed by an expert.

`make-dh-secret` *generator prime bit-length* [Procedure]

Generates a Diffie-Hellman secret key pair. Returns two values: the secret key (of bitwise length *bit-length*) part and the public key part.

`expt-mod` *base exponent modulus* [Procedure]

Computes `(mod (expt` *base exponent*`)` *modulus*`)`. This is modular exponentiation, so all the parameters must be integers.

The *exponent* can also be negative (set it to -1 to calculate the multiplicative inverse of *base*).

```
(import (industria crypto dh))
(let ((g modp-group15-g) (p modp-group15-p))
  (let-values (((y Y) (make-dh-secret g p 320))
               ((x X) (make-dh-secret g p 320)))
    ;; The numbers being compared are the shared secret
    (= (expt-mod X y modp-group15-p)
       (expt-mod Y x modp-group15-p))))
⇒ #t
```

This library also exports a few well known modular exponential (MODP) Diffie-Hellman groups (generators and primes) that have been defined by Internet RFCs. They are named modp-groupN-g (generator) and modp-groupN-p (prime) where N is the number of the group. Groups 1, 2, 5, 14, 15, 16, 17 and 18 are currently exported. They all have different lengths and longer primes are more secure but also slower. See RFC 3526 for more on this.

2.1.9 Digital Signature Algorithm

The `(industria crypto dsa)` library provides procedures for creating and verifying DSA signatures. DSA is a public key signature algorithm, which means that it uses private and public key pairs. With a private key you can create a signature that can then be verified by someone using the corresponding public key. The idea is that it's very difficult to create a correct signature without having access to the private key, so if the signature can be verified it must have been made by someone who has access to the private key.

DSA is standardized by FIPS Publication 186. It is available at this web site: `http://csrc.nist.gov/publications/PubsFIPS.html`.

There is currently no procedure to generate a new DSA key. Here is how to generate keys with OpenSSL or GnuTLS:

```
openssl dsaparam 1024 | openssl gendsa /dev/stdin > dsa.pem
certtool --dsa --bits 1024 -p > dsa.pem
```

The key can then be loaded with `dsa-private-key-from-pem-file`.

`make-dsa-public-key` *p q g y* [Procedure]
: Returns a DSA public key value. See the FIPS standard for a description of the parameters.

  To access the fields use `dsa-public-key-p`, `dsa-public-key-q`, `dsa-public-key-g` and `dsa-public-key-y`.

`dsa-public-key?` *obj* [Procedure]
: True if *obj* is a DSA public key value.

`dsa-public-key-length` *key* [Procedure]
: Returns the number of bits in the p value of *key*. This is often considered to be the length of the key. The bitwise-length of q is also important, it corresponds with the length of the hashes used for signatures.

`make-dsa-private-key` *p q g y x* [Procedure]
: Returns a DSA private key value. See the FIPS standard for a description of the parameters.

  To access the fields use `dsa-private-key-p`, `dsa-private-key-q`, `dsa-private-key-g`, `dsa-private-key-y` and `dsa-private-key-x`.

`dsa-private-key?` *obj* [Procedure]
: Returns `#t` if *obj* is a DSA private key.

`dsa-private->public` *private-key* [Procedure]
: Converts a private DSA key into a public DSA key by removing the private fields.

`dsa-private-key-from-bytevector` *bv* [Procedure]
: Parses *bv* as an ASN.1 DER encoded private DSA key.

`dsa-private-key-from-pem-file` *filename* [Procedure]
: Opens the file and reads a private DSA key. The file should be in Privacy Enhanced Mail (PEM) format and contain an ASN.1 DER encoded private DSA key.

  Encrypted keys are currently not supported.

`dsa-signature-from-bytevector` *bv* [Procedure]
Parses the bytevector *bv* as an ASN.1 DER encoded DSA signature. The return value is a list with the *r* and *s* values that make up a DSA signature.

`dsa-create-signature` *hash private-key* [Procedure]
The *hash* is the message digest (as a bytevector) of the data you want to sign. The *hash* and the *private-key* are used to create a signature which is returned as two values: `r` and `s`.

The *hash* can e.g. be an SHA-1 message digest. Such a digest is 160 bits and the *q* parameter should then be 160 bits.

`dsa-verify-signature` *hash public-key r s* [Procedure]
The *hash* is the message digest (as a bytevector) of the data which the signature is signing.

Returns `#t` if the signature matches, otherwise `#f`.

2.1.10 Elliptic Curve Cryptography

The `(industria crypto ec)` provides algorithms and definitions for working with elliptic curves.

Only curves over prime finite fields are currently supported. Points are either `+inf.0` (the point at infinity) or a pair of x and y coordinates.

Some standardized curves are exported:

`secp256r1`
This curve is equivalent to a 3072-bit RSA modulus.

`nistp256` Curve P-256. This is the same curve as above.

`secp384r1`
This curve is equivalent to a 7680-bit RSA modulus.

`nistp384` Curve P-384. This is the same curve as above.

`secp521r1`
This curve is equivalent to a 15360-bit RSA modulus. The "521" is not a typo.

`nistp521` Curve P-521. This is the same curve as above.

`make-elliptic-prime-curve` *p a b G n h* [Procedure]
Constructs a new elliptic-curve object given the domain parameters of a curve:

$y^2 \equiv x^3 + ax + b \pmod{p}$.

Normally one will be working with pre-defined curves, so this constructor can be safely ignored. The curve definition will include all these parameters.

`elliptic-prime-curve?` *obj* [Procedure]
Returns `#t` if *obj* is an elliptic prime curve.

The accessors can be safely ignored unless you're interested in the curves themselves.

`elliptic-curve-a` *elliptic-curve* [Procedure]
This is one of the parameters that defines the curve: an element in the field.

`elliptic-curve-b` *elliptic-curve* [Procedure]
This is one of the parameters that defines the curve: another element in the field.

`elliptic-curve-G` *elliptic-curve* [Procedure]
This is one of the parameters that defines the curve: the base point, i.e. an actual point on the curve.

`elliptic-curve-n` *elliptic-curve* [Procedure]
This is one of the parameters that defines the curve: a prime that is the order of G (the base point).

`elliptic-curve-h` *elliptic-curve* [Procedure]
This is one of the parameters that defines the curve: the cofactor.

`elliptic-prime-curve-p` *elliptic-prime-curve* [Procedure]
This is one of the parameters that defines the curve: the integer that defines the prime finite field.

`elliptic-curve=?` $elliptic\text{-}curve_1$ $elliptic\text{-}curve_2$ [Procedure]
Returns `#t` if the elliptic curve objects are equal (in the sense that all domain parameters are equal).

`ec+` *P Q elliptic-curve* [Procedure]
This adds the points *P* and *Q*, which must be points on *elliptic-curve*.

`ec-` *P* [*Q*] *elliptic-curve* [Procedure]
This subtracts *Q* from *P*, both of which must be points on *elliptic-curve*. If *Q* is omitted it returns the complement of *P*.

`ec*` *multiplier P elliptic-curve* [Procedure]
This multiplies *P* by *multiplier*. *P* must be a point on *elliptic-curve* and *multiplier* must be a non-negative integer.

This operation is the elliptic curve equivalence of `expt-mod`.

`bytevector->elliptic-point` *bytevector elliptic-curve* [Procedure]
Converts *bytevector* to a point on *elliptic-curve*. When points are sent over the network or stored in files they are first converted to bytevectors.

`integer->elliptic-point` *integer elliptic-curve* [Procedure]
Performs the same conversion as `bytevector->elliptic-point`, but first converts *integer* to a bytevector.

`->elliptic-point` *x elliptic-curve* [Procedure]
A generic procedure that accepts as input an *x* that is either already a point, a bytevector representing a point, or an integer representing a point.

`elliptic-point->bytevector` *point elliptic-curve* [Procedure]
Converts *point* to its bytevector representation. This representation is sometimes hashed, e.g. in SSH public keys, so the canonical representation is used for compatibility with other software.

2.1.11 Elliptic Curve Digital Signature Algorithm (ECDSA)

The `(industria crypto ecdsa)` library builds on the `(industria crypto ec)` library and provides an interface similar to `(industria crypto dsa)`. The keys and the operations are defined to work with elliptic curves instead of modular exponentiation.

`make-ecdsa-public-key` *elliptic-curve Q* [Procedure]
Constructs an ECDSA public key object. *Q* is a point on *elliptic-curve*. *Q* is only checked to be on the curve if it is in bytevector format.

`ecdsa-public-key?` *obj* [Procedure]
Returns `#t` if *obj* is an ECDSA public key object.

`ecdsa-public-key-curve` *ecdsa-public-key* [Procedure]
Returns the curve that *ecdsa-public-key* uses.

`ecdsa-public-key-Q` *ecdsa-public-key* [Procedure]
The point on the curve that defines *ecdsa-public-key*.

`ecdsa-public-key-length` *ecdsa-public-key* [Procedure]
The bitwise length of the ECDSA public key *ecdsa-public-key*.

`make-ecdsa-private-key` *elliptic-curve* [*d Q*] [Procedure]
Constructs an ECDSA private key object. *d* is a secret multiplier, which gives a public point *Q* on *elliptic-curve*.

If *Q* is omitted it is recomputed based on *d* and the curve. If *d* is omitted a random multiplier is chosen. Please note the warning about entropy at the start of this section. See Section 2.1 [crypto], page 2.

`ecdsa-private-key?` *obj* [Procedure]
Returns `#t` if *obj* is an ECDSA private key object.

`ecdsa-private-key-d` *ecdsa-private-key* [Procedure]
The secret multiplier of *ecdsa-private-key*.

`ecdsa-private-key-Q` *ecdsa-private-key* [Procedure]
The public point of *ecdsa-private-key*.

`ecdsa-private->public` *ecdsa-private-key* [Procedure]
Strips *ecdsa-private-key* of the secret multiplier and returns an ECDSA public key object.

`ecdsa-private-key-from-bytevector` *bytevector* [Procedure]
Parses *bytevector* as an ECDSA private key encoded in RFC 5915 format. A curve identifier is encoded along with the key. Currently only the curves secp256r1, secp384r1 and secp521r1 are supported.

`ecdsa-verify-signature` *hash ecdsa-public-key r s* [Procedure]
Returns `#t` if the signature (*r*,*s*) was made by the private key corresponding to *ecdsa-public-key*. The bytevector *hash* is the message digest that was signed.

`ecdsa-create-signature` *hash ecdsa-private-key* [Procedure]
Creates a signature of the bytevector *hash* using *ecdsa-private-key*. Returns the values r and s.

ECDSA keys are normally defined to work together with some particular message digest algorithm. RFC 5656 defines ECDSA with SHA-2 and this library provides the record types `ecdsa-sha-2-public-key` and `ecdsa-sha-2-private-key` so that keys defined to work with SHA-2 can be distinguished from other keys. Keys of this type are still usable for operations that expect the normal ECDSA key types.

`ecdsa-signature-from-bytevector` *bv* [Procedure]
Parses the bytevector *bv* as an ASN.1 DER encoded ECDSA signature as per RFC 4492. The return value is a list with the *r* and *s* values that make up a ECDSA signature. These values should be passed to `ecdsa-verify-signature`.

`ecdsa-signature-to-bytevector` *r s* [Procedure]
Encodes the ECDSA signature *r* and *s* as a bytevector in the ASN.1 DER format given in RFC 4492.

`make-ecdsa-sha-2-public-key` *elliptic-curve Q* [Procedure]
Performs the same function as `make-ecdsa-public-key`, but the returned key is marked to be used with SHA-2.

`ecdsa-sha-2-public-key?` *obj* [Procedure]
Returns `#t` if *obj* is an ECDSA public key marked to be used with SHA-2.

`make-ecdsa-sha-2-private-key` [Procedure]
Performs the same function as `make-ecdsa-private-key`, but the returned key is marked to be used with SHA-2.

`ecdsa-sha-2-private-key?` [Procedure]
Returns `#t` if *obj* is an ECDSA private key marked to be used with SHA-2.

`ecdsa-sha-2-verify-signature` *message ecdsa-sha2-public-key r s* [Procedure]
The bytevector *message* is hashed with the appropriate message digest algorithm (see RFC 5656) and the signature (*r*,*s*) is then verified. Returns `#t` if the signature was made with the private key corresponding to *ecdsa-sha2-public-key*.

`ecdsa-sha-2-create-signature` *message ecdsa-sha2-private-key* [Procedure]
The bytevector *message* is hashed with the appropriate message digest algorithm (see RFC 5656) and a signature is created using *ecdsa-sha2-private-key*. Returns the values r and s.

`ecdsa-sha-2-private-key-from-bytevector` *bytevector* [Procedure]
Performs the same function as `ecdsa-private-key-from-bytevector`, except the returned value is marked to be used with SHA-2.

2.1.12 Entropy and randomness

The `(industria crypto entropy)` library is meant to help with generating random data. It tries to use the system's `/dev/urandom` device if possible, otherwise it uses SRFI-27.

Please see the note at the beginning of the chapter.

`bytevector-randomize!` *target* [*target-start k*] [Procedure]
Writes *k* random bytes to the bytevector *target* starting at index *target-start*.

`make-random-bytevector` *k* [Procedure]
Returns a bytevector of length *k* with random content.

```
(import (industria crypto entropy))
(make-random-bytevector 8)
⇒ #vu8(68 229 38 253 58 70 198 161)
```

2.1.13 RSA public key encryption and signatures

The `(industria crypto rsa)` library implements the RSA (Rivest, Shamir and Adleman) algorithm and a few helpers.

`make-rsa-public-key` *n e* [Procedure]
Returns an RSA public key object containing the modulus *n* and the public exponent *e*.

`rsa-public-key?` *obj* [Procedure]
True if *obj* is a public RSA key.

`rsa-public-key-n` *key* [Procedure]

`rsa-public-key-modulus` *key* [Procedure]
Returns the *modulus* of *key*.

`rsa-public-key-e` *key* [Procedure]

`rsa-public-key-public-exponent` *key* [Procedure]
Returns the *public exponent* of *key*.

`rsa-public-key-from-bytevector` *bytevector* [Procedure]
Parses *bytevector* as an ASN.1 DER encoded public RSA key. The return value can be used with the other procedures in this library.

`rsa-public-key-length` *key* [Procedure]
Returns the number of bits in the modulus of *key*. This is also the maximum length of data that can be encrypted or decrypted with the key.

`rsa-public-key-byte-length` *key* [Procedure]
Returns the number of 8-bit bytes required to store the modulus of *key*.

`make-rsa-private-key` *n e d* [*p q exponent1 exponent2 coefficient*] [Procedure]
Returns an RSA private key object with the given modulus *n*, public exponent *e*, and private exponent *d*.

The other parameters are used to improve the efficiency of `rsa-encrypt`. They are optional and will be computed if they are omitted.

`rsa-private-key?` *obj* [Procedure]
: True if *obj* is a private RSA key.

`rsa-private-key-n` *key* [Procedure]

`rsa-private-key-modulus` *key* [Procedure]
: Returns the *modulus* of *key*.

`rsa-private-key-public-exponent` *key* [Procedure]
: Returns the *public exponent* of *key*. This exponent is used for encryption and signature verification.

`rsa-private-key-d` *key* [Procedure]

`rsa-private-key-private-exponent` *key* [Procedure]
: Returns the *private exponent* of *key*. This exponent is used for decryption and signature creation.

`rsa-private-key-prime1` *key* [Procedure]

`rsa-private-key-prime2` *key* [Procedure]
: These two procedures return the first and second prime factors (p,q) of the modulus ($n = pq$).

`rsa-private-key-exponent1` *key* [Procedure]
: This should be equivalent to `(mod d (- p 1))`. It is used to speed up `rsa-decrypt`.

`rsa-private-key-exponent2` *key* [Procedure]
: This should be equivalent to `(mod d (- q 1))`. It is used to speed up `rsa-decrypt`.

`rsa-private-key-coefficient` *key* [Procedure]
: This should be equivalent to `(expt-mod q -1 p)`. It is used to speed up `rsa-decrypt`.

`rsa-private->public` *key* [Procedure]
: Uses the *modulus* and *public exponent* of *key* to construct a public RSA key object.

`rsa-private-key-from-bytevector` *bytevector* [Procedure]
: Parses *bytevector* as an ASN.1 DER encoded private RSA key. The return value can be used with the other procedures in this library.

`rsa-private-key-from-pem-file` *filename* [Procedure]
: Opens the file and reads a private RSA key. The file should be in Privacy Enhanced Mail (PEM) format and contain an ASN.1 DER encoded private RSA key.

  Encrypted keys are currently not supported.

`rsa-encrypt` *plaintext key* [Procedure]
: Encrypts the *plaintext* integer using the *key*, which is either a public or private RSA key.

  plaintext must be an exact integer that is less than the modulus of *key*.

`rsa-decrypt` *ciphertext key* [Procedure]

Decrypts the *ciphertext* integer using the *key*, which must be a private RSA key.

ciphertext must be an exact integer that is less than the modulus of *key*.

```
(import (industria crypto rsa))
(let ((key (make-rsa-private-key 3233 17 2753)))
  (rsa-decrypt (rsa-encrypt 42 key) key))
⇒ 42
```

`rsa-decrypt/blinding` *ciphertext key* [Procedure]

This performs the same function as `rsa-decrypt`, but it uses RSA blinding. It has been shown that the private key can be recovered by measuring the time it takes to run the RSA decryption function. Use RSA blinding to protect against these timing attacks.

For more technical information on the subject, see Paul C. Kocher's article Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems (`http://www.cryptography.com/resources/whitepapers/TimingAttacks.pdf`).

It is often not enough to just use the plain encryption and decryption procedures; a protocol for what to put in the plaintext should also be used. PKCS #1 (RFC 3447) is a standard for how to perform RSA encryption and signing with padding. New protocols should use one of the other protocols from the RFC.

`rsa-pkcs1-encrypt` *plaintext public-key* [Procedure]

Pads and encrypts the *plaintext* bytevector using *public-key*, a public RSA key. The return value is an integer.

The plaintext can't be longer than the length of the key modulus, in bytes, minus 11.

`rsa-pkcs1-decrypt` *ciphertext private-key* [Procedure]

The inverse of *rsa-pkcs1-encrypt*. Decrypts the *ciphertext* integer using *private-key*, a private RSA key. The padding is then checked for correctness and removed.

```
(import (industria crypto rsa))
(let ((key (make-rsa-private-key
            28841272834746329365019147630367075358 3
            65537
            19090504838050197105561255893672549699 3)))
  (utf8->string
   (rsa-pkcs1-decrypt
    (rsa-pkcs1-encrypt (string->utf8 "Hello")
                       (rsa-private->public key))
    key)))
⇒ "Hello"
```

`rsa-pkcs1-decrypt-signature` *signature public-key* [Procedure]

Decrypts the signature (a bytevector) contained in the *signature* integer by using the *public-key*. The signature initially contains PKCS #1 padding, but this is removed.

`rsa-pkcs1-encrypt-signature` *digest private-key* [Procedure]

Sign the bytevector *digest* using *private-key*. The inverse of `rsa-pkcs1-decrypt-signature`.

`rsa-pkcs1-decrypt-digest` *signature public-key* [Procedure]
: This performs the same operation as `rsa-pkcs1-decrypt-signature`, except it then treats the decrypted signature as a DER encoded DigestInfo. The return value is a list containing a digest algorithm specifier and a digest.

`rsa-pkcs1-encrypt-digest` *algorithm digest private-key* [Procedure]
: Create a DER encoded DigestInfo signature (inverse of `rsa-pkcs1-decrypt-digest`). The *digest* must be a bytevector and should have a length appropriate for the *algorithm*, which may be either an object ID or one of these symbols: md5, sha-1, sha-224, sha-256, sha-384, sha-512, sha-512-224, sha-512-256, sha3-224, sha3-256, sha3-384, sha3-512, shake-128, shake-256.

2.2 OpenPGP signature verification

The `(industria openpgp)` library provides procedures for reading OpenPGP keyrings and verifying signatures. OpenPGP signatures can be created with e.g. GNU Private Guard (GnuPG) and are often used to verify the integrity of software releases.

Version 4 keys and version 3/4 signatures are supported. The implemented public key algorithms are RSA and DSA, and it verifies signatures made using the message digest algorithms MD5, SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512 (all the standard algorithms except RIPE-MD160).

An OpenPGP key is actually a list of OpenPGP packets with a certain structure: first is the primary key (e.g. an RSA or DSA key), next possibly a revocation, then a number of user IDs, attributes, signatures and also subkeys (which are just like primary keys, except marked as subkeys). See RFC 4880 section 11 for the exact composition. This library represents keyrings as hashtables indexed by key ID and where the entries are lists of packets in the order they appeared in the keyring file.

Please note that this library assumes the program that wrote the keyring did due diligence when importing keys, and made sure that e.g. subkey binding signatures are verified, and that the order of packets is correct.

`port-ascii-armored?` *port* [Procedure]
: Returns false if the data at the beginning of *port* doesn't look like a valid binary OpenPGP packet. The port must be a binary input port. The port position is not changed.

`get-openpgp-packet` *port* [Procedure]
: Reads an OpenPGP packet from *port*, which must be a binary input port. An error is raised if the packet type is unimplemented.

`get-openpgp-keyring` *p* [Procedure]
: Reads a keyring from the binary input port *p*. Returns a hashtable where all primary keys and subkeys are indexed by their key ID (an integer). The values in the hashtable are lists that contain all OpenPGP packets associated with each key. No effort at all is made to verify that keys have valid signatures.

  Warning: this can take a while if the keyring is very big.

`get-openpgp-keyring/keyid` *p keyid* [Procedure]
Searches the binary input port *p* for the public key with the given *keyid*. Returns a hashtable similar to `get-openpgp-keyring`, except it will only contain the primary and subkeys associated with the *keyid*.

The *keyid* can be either a 64 or 32 bit exact integer.

Warning: this is faster than get-openpgp-keyring, but is still rather slow with big keyrings. The speed depends on the SHA-1 implementation.

`get-openpgp-detached-signature/ascii` *p* [Procedure]
Reads a detached OpenPGP signature from the textual input port *p*. Returns either an OpenPGP signature object or the end of file object.

These signatures can be created with e.g. `gpg -a --detach-sign filename`.

`verify-openpgp-signature` *sig keyring p* [Procedure]
Verifies the signature data in *sig*. The *keyring* hashtable is used to find the public key of the signature issuer. The signed data is read from the binary input port *p*.

This procedure returns two values. These are the possible combinations:

- `good-signature` *key-data* – The signature matches the data. The *key-data* contains the public key list that was used to verify the signature.
- `bad-signature` *key-data* – The signature does not match the data. The *key-data* is the same as above.
- `missing-key` *key-id* – The issuer public key for the signature was not found in the keyring. The *key-id* is the 64-bit key ID of the issuer.

`openpgp-signature?` *obj* [Procedure]
True if *obj* is an OpenPGP signature object. Such objects are read with `get-openpgp-detached-signature/ascii` and are also contained in keyring entries.

`openpgp-signature-issuer` *sig* [Procedure]
The 64-bit key ID of the OpenPGP public key that issued the signature *sig*.

`openpgp-signature-public-key-algorithm` *sig* [Procedure]
Returns the name of the public key algorithm used to create the signature *sig*. This is currently the symbol `dsa` or *rsa*.

`openpgp-signature-hash-algorithm` *sig* [Procedure]
The name of the message digest algorithm used to create the signature *sig*. This is currently one of `md5`, `sha-1`, `ripe-md160` (unsupported), `sha-224`, `sha-256`, `sha-384` or `sha-512`.

`openpgp-signature-creation-time` *sig* [Procedure]
An SRFI-19 date object representing the time at which the signature *sig* was created.

`openpgp-signature-expiration-time` *sig* [Procedure]
An SRFI-19 date object representing the time at which the signature *sig* expires. Returns `#f` if there's no expiration time.

`openpgp-user-id?` *obj* [Procedure]
True if *obj* is an OpenPGP user id.

`openpgp-user-id-value` *user-id* [Procedure]
The string value of the *user-id*. This is often the name of the person who owns the key.

`openpgp-user-attribute?` *obj* [Procedure]
True if *obj* is an OpenPGP user attribute. Attributes are used to encode JPEG images. There's currently no way to access the image.

`openpgp-public-key?` *obj* [Procedure]
True if *obj* is an OpenPGP primary key or subkey.

`openpgp-public-key-subkey?` *key* [Procedure]
True if *obj* is a subkey.

`openpgp-public-key-value` *key* [Procedure]
The DSA or RSA public key contained in the OpenPGP public *key*. The value returned has the same type as the `(crypto weinholt dsa)` or `(crypto weinholt rsa)`.

`openpgp-public-key-fingerprint` *key* [Procedure]
The fingerprint of the OpenPGP public *key* as a bytevector. This is an SHA-1 digest based on the public key values.

`openpgp-format-fingerprint` *bv* [Procedure]
Formats the bytevector *bv*, which was presumably created by `openpgp-public-key-fingerprint`, as a string in the format preferred for PGP public key fingerprints.

`openpgp-public-key-id` *key* [Procedure]
The 64-bit key ID of the OpenPGP public *key*.

2.3 Off-the-Record Messaging

The `(industria otr)` library provides Off-the-Record Messaging (OTR), which is a security protocol for private chat. It can be tunneled over any protocol that guarantees in-order delivery (e.g. IRC or XMPP). It provides encryption, authentication, deniability and perfect forward secrecy.

This library does not manage user identities, which is something the OTR Development Team's C library does. This choice was made to keep the implementation simple and focused on the protocol only.

The website for OTR is `http://www.cypherpunks.ca/otr/`.

`otr-message?` *str* [Procedure]
Returns `#t` if *str*, which is a message from a remote party, contains an OTR message. If it is an OTR message you should look up the OTR state that corresponds to the remote party (possibly make a new state) and call `otr-update!`.

`make-otr-state` *dsa-key mss* [*instance-tag* [*versions*]] [Procedure]
Creates an OTR state value given the private DSA key *dsa-key* and a maximum segment size *mss*. The state is used to keep track of session keys and incoming message fragments.

The *dsa-key* must have a 160-bit q-parameter because of details in the protocol and limitations of other implementations. A 1024-bit DSA key will work. See Section 2.1.9 [crypto dsa], page 11.

The maximum segment size *mss* is used to split long OTR messages into smaller parts when OTR is used over a protocol with a maximum message size, e.g. IRC.

If an *instance-tag* is specified it must be a 32-bit integer not less than `#x100`. If it is omitted or `#f` an instance tag will be randomly generated. OTR version 3 uses the instance tags to identify which OTR state messages belongs to. Be sure to read the documentation for `otr-state-our-instance-tag`. New for Industria 1.5.

If *versions* is not omitted it must be a list of acceptable OTR protocol versions. The default is `(2 3)`. New for Industria 1.5.

`otr-update!` *state str* [Procedure]

Processes the *str* message, which came from the remote party, and updates the *state*. Use `otr-empty-queue!` to retrieve scheduled events.

`otr-send-encrypted!` *state msg* [Procedure]

This is used to send a message to the remote party. It encrypts and enqueues the *msg* bytevector and updates the *state*. Use `otr-empty-queue!` to retrieve the encrypted and formatted messages that should be sent to the remote party.

The *msg* must not contain a NUL (0) byte.

`otr-authenticate!` *state secret* [*question*] [Procedure]

Initiate or respond to an authentication request. After calling this procedure you should use `otr-empty-queue!`, just like with `otr-send-encrypted!`.

The authentication protocol can be used to verify that both partyies know the *secret* bytevector. The secret is never revealed over the network and is not even transmitted in an encrypted form. The protocol used is the Socialist Millionaires' Protocol (SMP), which is based on a series of zero-knowledge proofs.

`otr-empty-queue!` *state* [Procedure]

Returns and clears the event queue. The queue is a list of pairs where the symbol in the `car` of the pair determines its meaning. These are the possible types:

- `(outgoing .` *line*`)` – The `cdr` is a string that should be sent to the remote party.
- `(encrypted .` *msg*`)` – The `cdr` is a string that contains a decrypted message that was sent by the remote party.
- `(unencrypted .` *msg*`)` – The `cdr` is a string that was sent *unencrypted* by the remote party. This happens when a whitespace-tagged message is received.
- `(session-established .` *whence*`)` – A session has been established with the remote party. It is now safe to call `otr-state-their-dsa-key`, `otr-state-secure-session-id`, `otr-send-encrypted!` and `otr-authenticate!`. The `cdr` is the symbol `from-there` if the session was initiated by the remote party. Otherwise it is `from-here`.
- `(session-finished .` *whom*`)` – The session is now finished and no new messages can be sent over it. The `cdr` is either the symbol `by-them` or `by-us`. *Note*: there is currently no way to finish the session from the local side, so `by-us` is not used yet.

- `(authentication . expecting-secret)` – The remote party has started the authentication protocol and now expects you to call `otr-authenticate!`.
- `(authentication . #t)` – The authentication protocol has succeeded and both parties had the same secret.
- `(authentication . #f)` – The authentication protocol has failed. The secrets were not identical.
- `(authentication . aborted-by-them)` – The remote party has aborted the authentication protocol.
- `(authentication . aborted-by-us)` – The local party has encountered an error and therefore aborted the authentication protocol.
- `(they-revealed .` *k*`)` – The remote party revealed an old signing key. This is a normal part of the protocol and the key is sent unencrypted to ensure the deniability property. You might like to reveal the key somehow yourself in case you're tunneling OTR over an encrypted protocol.
- `(we-revealed .` *k*`)` – The local party has revealed an old signing key. *Note*: currently not used.
- `(undecipherable-message . #f)` – An encrypted message was received, but it was not possible to decrypt it. This might mean e.g. that the remote and local parties have different sessions or that a message was sent out of order.
- `(remote-error .` *msg*`)` – The remote party encountered a protocol error and sent a plaintext error message (probably in English).
- `(local-error .` *con*`)` – There was an exception raised during processing of a message. The `cdr` is the condition object.
- `(symmetric-key-request . (`*protocol* `.` *data*`))` – The remote party has requested that the extra symmetric key be used to communicate in some out-of-band protocol. See `otr-send-symmetric-key-request!`. New for Industria 1.5.

For forward-compatibility you should ignore any pair with an unknown `car`. Most messages are quite safe to ignore if you don't want to handle them.

`otr-state-their-dsa-key` *state* [Procedure]
Returns the remote party's public DSA key. This should be used to verify the remote party's identity. If the SMP authentication protocol succeeds you can remember the hash of the key for the next session. The user could also verify the key's hash by cell phone telephone or something.

`otr-state-our-dsa-key` *state* [Procedure]
Returns the local party's private DSA key. This is useful when the user is on the phone with the remote party. First convert it to a public key with `dsa-private->public` and then hash it with `otr-hash-public-key`.

`otr-hash-public-key` *public-dsa-key* [Procedure]
Hashes a public DSA key and formats it so that it can be shown to the OTR user.

`otr-state-secure-session-id` *state* [Procedure]
Returns the *secure session ID* associated with the OTR state.

`otr-format-session-id` *id* [Procedure]
Formats a secure session ID in the format that is recommended when the ID should be shown to the OTR user.

The first part of the ID should be shown in bold if the session was initiated by the local party. Otherwise the second part should be bold.

`otr-state-version` *state* [Procedure]
The OTR protocol version used by the state. This is either the integer `2` or the integer `3`. New for Industria 1.5.

`otr-state-mss` *state* [Procedure]
Returns the current maximum segment size of the OTR state.

`otr-state-mss-set!` *state int* [Procedure]
Sets *int* as the maximum segment size of the OTR state.

OTR protocol version 3 defines an extra symmetric key.

`otr-send-symmetric-key-request!` *state protocol data* [Procedure]
This sends a message to the remote party that requests that it uses the extra symmetric key for some out-of-band protocol.

The remote party may ignore this request if the OTR protocol version (as returned by `otr-state-version`) is not at least 3.

The *protocol* parameter is an unsigned 32-bit integer that indicates what the key should be used for. At the time this manual is written there are no defined uses. One might expect a list of uses to appear in the protocol documentation at `http://www.cypherpunks.ca/otr/`.

The *data* parameter is a bytevector containing protocol-dependent data.

`otr-state-symmetric-key` *state* [Procedure]
This returns the extra symmetric key in the form of a 256-bit bytevector.

`otr-tag` *whitespace? versions* [Procedure]
Constructs a string that may be sent to a remote party as a request to start an OTR session. New for Industria 1.5.

If *whitespace?* is true then a whitespace tag will be made. This tag may be appended to a normal message sent by the user. If the recipient's client supports OTR it may start a session, but if it does not support OTR then hopefully it will not show the whitespaces.

The *versions* argument specifies which OTR protocol versions should be present in the tag. This can either be a list of version numbers or the symbol `all`.

`otr-state-our-instance-tag` *state* [Procedure]
This returns the local instance tag. It is new for Industria 1.5.

It is intended for instance tags to be persistent across client restarts. If the local party crashes then the remote party may still have an OTR session established. If the local client were then to change its instance tag on restart it would not receive any messages from the remote party and would not send error messages. To the remote party it would look like they were being ignored.

Isn't this the most boring manual you've ever read?

Version history:

- Industria 1.5 introduced support for protocol version 3. This new version of the protocol uses instance tags, which are used to distinguish between different OTR sessions. This fixes a problem with chat networks that allow multiple logins. The new version also defines an extra symmetrical key that can be used by out-of-band protocols.

2.4 Secure Shell (SSH)

The `(industria ssh)` library hierarchy deals with the Secure Shell protocol. Both SSH servers and clients can be written with these libraries. Some convenient abstractions are currently missing though, e.g. a channel abstraction. These libraries hide the details of the wire protocol and the cryptographic algorithms. The protocol is standardized by a series of RFCs: 4250, 4251, 4252, 4253, 4254, etc.

No TCP server abstraction is provided by Industria. To make a server you will probably need to use your implementation's network abstractions.

It remains to be seen if this interface can be used for interactive applications. One problem is `get-ssh`, which reads a whole SSH packet. This procedure is blocking. R6RS doesn't provide any procedures for event-driven programming, so the author has made no effort to make this library work in an event-driven setting.

`ssh-debugging` [Parameter]
: This SRFI-39 parameter controls debug output. It is a bit field with three bits currently defined. Bit 0 enables general trace messages, bit 1 enables packet traces and bit 2 enables packet hexdumps.

  Default: `#b000`

`ssh-debugging-port` [Parameter]
: This SRFI-39 parameter controls where debug output is written to. It defaults to the error port that was current when the library top-level was run.

`identification-protocol-version` [Parameter]
: This SRFI-39 parameter is used when constructing the local identification string. It specifies which SSH protocol version number is supported.

  Default: `"2.0"`

`identification-software-version` [Parameter]
: This SRFI-39 parameter is used when constructing the local identification string. It specifies the name and version of the client or server.

  Default: `"Industria_2"`

`identification-comments` [Parameter]
: This SRFI-39 parameter is used when constructing the local identification string. It is `#f` or optionally a string of comments. This field is sometimes used to identify a vendor.

  Default: `#f`

The following parameters are when constructing the local kex exchange packet. It lists the preferred algorithms. You may remove and reorder the algorithms, but you can't introduce new ones without first adding them to `(industria ssh algorithms)`. The defaults may change in the future.

`preferred-kex-algorithms` [Parameter]
This is a list of key exchange algorithm names in the order they are preferred.

Default: ("curve25519-sha256" "curve25519-sha256@libssh.org" "diffie-hellman-group-exchange-sha256" "diffie-hellman-group-exchange-sha1" "diffie-hellman-group14-sha1" "diffie-hellman-group1-sha1")

`preferred-server-host-key-algorithms` [Parameter]
This is a list of host key algorithm names in the order they are preferred. The server may have more than one host key and this is used to decide between them.

Default: ("rsa-sha2-512" "rsa-sha2-256" "ssh-rsa" "ecdsa-sha2-nistp256" "ecdsa-sha2-nistp384" "ecdsa-sha2-nistp521" "ssh-ed25519" "ssh-dss")

`preferred-encryption-algorithms-client->server` [Parameter]
This is a list of encryption algorithm names in the order they are preferred for communication from the client to the server.

Default: ("aes128-ctr" "aes192-ctr" "aes256-ctr" "aes128-cbc" "aes192-cbc" "aes256-cbc" "blowfish-cbc" "arcfour256" "arcfour128" "3des-cbc")

`preferred-encryption-algorithms-server->client` [Parameter]
This is a list of encryption algorithm names in the order they are preferred for communication from the server to the client.

Default: ("aes128-ctr" "aes192-ctr" "aes256-ctr" "aes128-cbc" "aes192-cbc" "aes256-cbc" "blowfish-cbc" "arcfour256" "arcfour128" "3des-cbc")

`preferred-mac-algorithms-client->server` [Parameter]
This is a list of message authentication code algorithms in the order they are preferred for communication from the client to the server.

Default: ("hmac-md5" "hmac-sha1" "hmac-sha1-96" "hmac-md5-96")

`preferred-mac-algorithms-server->client` [Parameter]
This is a list of message authentication code algorithms in the order they are preferred for communication from the server to the client.

Default: ("hmac-md5" "hmac-sha1" "hmac-sha1-96" "hmac-md5-96")

`preferred-compression-algorithms-client->server` [Parameter]
This is a list of compression algorithms for packets transmitted from the client to the server.

Default: ("none")

`preferred-compression-algorithms-server->client` [Parameter]
This is a list of compression algorithms for packets transmitted from the server to the client.

Default: ("none")

`preferred-languages-client->server` [Parameter]
This is currently not used.

Default: ()

`preferred-languages-server->client` [Parameter]
This is currently not used.

Default: ()

`make-ssh-client` *binary-input-port binary-output-port* [Procedure]
Starts an SSH client connection over the two given ports, which should be connected to a server via TCP (or some other similar means).

If everything goes right an `ssh-conn` object is returned. The *peer identification* and *kexinit* fields are valid.

`make-ssh-server` *binary-input-port binary-output-port keys* [Procedure]
Starts an SSH server connection over the two given ports, which should be connected to a client via TCP (or some other similar means).

keys is a list of host keys. The currently supported key types are `dsa-private-key` and `ecdsa-sha-2-private-key`.

If everything goes right an `ssh-conn` object is returned. The *peer identification* and *kexinit* fields are valid.

`ssh-key-exchange` *ssh-conn* [Procedure]
This runs the negotiated key exchange algorithm on *ssh-conn*. After this is done the client will have received one of the server's public keys. The negotiated encryption and MAC algorithms will have been activated.

`ssh-conn-peer-identification` *ssh-conn* [Procedure]
The identification string the peer sent. This is a string that contains the peer's protocol version, software version and optionally some comments.

`ssh-conn-peer-kexinit` *ssh-conn* [Procedure]
This is the peer's key exchange initialization (kexinit) packet. It lists the peer's supported algorithms. See Section 2.4.2 [ssh transport], page 39.

`ssh-conn-host-key` *ssh-conn* [Procedure]
The server's public key. This has unspecified contents before the `ssh-key-exchange` procedure returns.

`ssh-conn-session-id` *ssh-conn* [Procedure]
The session ID of *ssh-conn*. This has unspecified contents before the `ssh-key-exchange` procedure returns.

`ssh-conn-registrar` *ssh-conn* [Procedure]
Returns a procedure that can be used to register parsers and formatters for SSH packet types. The returned procedure should be given as an argument to `register-connection` and `register-userauth`.

`ssh-error` *ssh-conn who message code irritants ...* [Procedure]
Sends a `disconnect` packet to the peer. The packet contains the message and the code. The connection is then closed and an error is raised.

The error code constants are defined elsewhere. See Section 2.4.2 [ssh transport], page 39.

`put-ssh` *ssh-conn pkt* [Procedure]
Sends the SSH packet *pkt* to the peer of *ssh-conn*.

`get-ssh` *ssh-conn* [Procedure]
Reads an SSH packet object from the peer of *ssh-conn*. The end-of-file object will be returned if the peer has closed the connection. The procedure blocks until a message has been received. Any messages of the type `ignore` are ignored.

Packet types must be registered before they can be received. Initially only the transport layer types are registered. If an unregistered type is received this procedure returns a list of two items: the symbol `unimplemented` and the unparsed contents of the packet. A packet of type *unimplemented* is sent to the peer.

`close-ssh` *ssh-conn* [Procedure]
Flushes the output port of *ssh-conn*, and then closes both the input and output ports.

`flush-ssh-output` *ssh-conn* [Procedure]
Flushes any pending output on *ssh-conn*.

The procedures below are used in the implementation of key re-exchange. After the initial key exchange either party can initiate a key re-exchange. RFC 4253 has the following to say on the subject:

> It is RECOMMENDED that the keys be changed after each gigabyte of transmitted data or after each hour of connection time, whichever comes sooner. However, since the re-exchange is a public key operation, it requires a fair amount of processing power and should not be performed too often.

The demonstration program `secsh-client` contains an example of how to initiate key re-exchange. The server demonstration program `honingsburk` also handles key re-exchange, but does not initiate it. See Section 3.2 [honingsburk], page 53.

`build-kexinit-packet` *ssh-conn* [Procedure]
Constructs and returns a key exchange packet for use by the local side.

`key-exchange-packet?` *pkt* [Procedure]
Returns `#t` if *pkt* should be given to `process-key-exchange-packet` for handling by the key exchange logic.

`ssh-key-re-exchange` *ssh-conn peer-kex local-kex* [Procedure]
Initiates key re-exchange on *ssh-conn*. This requires the peer's key exchange packet *peer-kex*, and the local key exchange packet *local-kex*. The procedure returns before the key re-exchange is finished. Both sides of the algorithm will need to communicate to complete the exchange.

`process-key-exchange-packet` *ssh-conn pkt* [Procedure]

Updates the key exchange logic on *ssh-conn* with the contents of *pkt*. If the packet is a `kexinit` packet and *ssh-conn* is a server, then this will automatically initiate the key re-exchange algorithm.

The procedure may return the symbol `finished` to indicate that the key exchange algorithm has finished and the new algorithms are used for packets sent to the peer.

Note: This interface is currently balanced in favor of servers. More experience in using the library is needed to determine how to make the key re-exchange interface better for clients. Suggestions are welcome.

2.4.1 Secure Shell Connection Protocol

The `(industria ssh connection)` library implements record types, parsers and formatters for the connection protocol packets in SSH.

The connection protocol handles two types of communication: global requests and channels. The global requests can be used to setup TCP/IP port forwarding. Most communication over SSH passes through channels. Channels are opened with the `channel-open` requests. The client and the server each assign an ID number to a channel: one ID is sent in the `channel-open` packet, the other ID in the `channel-open-confirmation` packet. In Industria all packets that are directed to a specific channel inherit from the `channel-packet` record type and the ID can be found with the `channel-packet-recipient` procedure.

Strings and bytevectors may be used interchangeably when constructing packets. Strings will automatically be converted with `string->utf8`. When these packets are received the parser will either parse those fields either as a string or a bytevector. A bytevector will be used when the field can contain more or less arbitrary data, e.g. filenames.

The text of this section uses the words “packet”, “message” and “request” interchangeably.

See RFC 4254 for a more detailed description of this protocol.

`register-connection` *registrar* [Procedure]

Registers the packet types for the connection protocol so that they may be received and sent. A registrar may be obtained from an ssh-conn object using `ssh-conn-registrar`.

`make-global-request` *type want-reply?* [Procedure]

Constructs a global request: a connection request not related to any channel. Some global requests contain additional fields. These requests are represented by the `global-request/*` packets.

`global-request?` *obj* [Procedure]

Returns true if *obj* is a `global-request?` packet.

`global-request-type` *pkt* [Procedure]

This field contains a string identifying the type of the request, e.g. `"no-more-sessions@openssh.com"`.

`global-request-want-reply?` *pkt* [Procedure]

This field is true if the sender expects a `request-success` or `request-failure` record in response.

`make-global-request/tcpip-forward` *want-reply? address port* [Procedure]
Constructs a request that instructs the server to bind a TCP server port and forward connections to the client.

`global-request/tcpip-forward?` *obj* [Procedure]
Returns true if *obj* is a `global-request/tcpip-forward` packet.

`global-request/tcpip-forward-address` *req* [Procedure]
This field is a string that represents the address to which the server should bind the TCP server port. Some addresses are given special meaning:

`""` The server should listen to all its addresses on all supported protocols (IPv4, IPV6, etc).

`"0.0.0.0"`
The server should listen to all its IPv4 addresses.

`"::"` The server should listen to all its IPv6 addresses.

`"localhost"`
The server should listen to its loopback addresses on all supported protocols.

`"127.0.0.1"`
The server should listen to its IPv4 loopback address.

`"::1"` The server should listen to its IPv6 loopback address.

`global-request/tcpip-forward-port` *req* [Procedure]
This field is an integer representing the port number to which the server should bind the TCP server port. If the number is `0` and *want-reply?* is true, the server will pick a port number and send it to the client in a `request-success` packet (the port number can be recovered with `(unpack "!L" (request-success-data response))`).

`make-global-request/cancel-tcpip-forward` *want-reply? address port* [Procedure]
Constructs a message that undoes the effect of a `global-request/tcpip-forward` request.

`global-request/cancel-tcpip-forward?` *obj* [Procedure]
Returns true if *obj* is a `global-request/cancel-tcpip-forward` packet.

`global-request/cancel-tcpip-forward-address` *req* [Procedure]
See `global-request/tcpip-forward-address`.

`global-request/cancel-tcpip-forward-port` *req* [Procedure]
See `global-request/tcpip-forward-port`.

`make-request-success` *data* [Procedure]
Constructs a packet which indicates that the previous `global-request` was successful.

`request-success?` *obj* [Procedure]
Returns true if *obj* is a `request-success` packet.

`request-success-data` *pkt* [Procedure]
This field contains a request-specific bytevector which is mostly empty.

`make-request-failure` [Procedure]
Returns an object which indicates that a global request failed.

`request-failure?` *obj* [Procedure]
Returns true if *obj* is a `request-failure` packet.

All requests to open a channel are represented by `channel-open/*` packets.

`channel-open?` *obj* [Procedure]
Returns true if *obj* is a `channel-open` packet.

`channel-open-type` *pkt* [Procedure]
A string representing the type of the `channel-open` request, e.g. `"session"`.

`channel-open-sender` *pkt* [Procedure]
This is the ID for the sender side of the channel.

`channel-open-initial-window-size` *pkt* [Procedure]
This is the window size of the channel. The window size is used for flow-control and it decreases when data is sent over the channel and increases when a `channel-window-adjust` packet is sent. Each side of a channel has a window size.

`channel-open-maximum-packet-size` *pkt* [Procedure]
This is the maximum allowed packet size for data sent to a channel. It basically limits the size of `channel-data` and `channel-extended-data` packets.

`make-channel-open/direct-tcpip` *sender-id initial-window-size connect-address connect-port originator-address originator-port* [Procedure]
Constructs a request to open a new channel which is then connected to a TCP port.

`channel-open/direct-tcpip?` *obj* [Procedure]
Returns true if *obj* is a `channel-open/direct-tcpip` packet.

`channel-open/direct-tcpip-connect-address` *pkt* [Procedure]
This is the hostname or network address that the TCP connection should be connected to.

`channel-open/direct-tcpip-connect-port` *pkt* [Procedure]
This is the port number that the TCP connection should be connected to.

`channel-open/direct-tcpip-originator-address` *pkt* [Procedure]
This is the network address of the machine that made the request.

`channel-open/direct-tcpip-originator-port` *pkt* [Procedure]
This is the port number on which the request was made. This is useful when a client implements forwarding of client-local TCP ports.

`make-channel-open/forwarded-tcpip` *sender-id initial-window-size maximum-packet-size connected-address connected-port originator-address originator-port* [Procedure]

This request is used by the server to tell the client that a TCP connection has been requested to a port for which the client sent a `global-request/tcpip-forward` request.

`channel-open/forwarded-tcpip?` *obj* [Procedure]

Returns true if *obj* is a `channel-open/forwarded-tcpip` packet.

`channel-open/forwarded-tcpip-connected-address` *pkt* [Procedure]

The address to which the TCP connection was made.

`channel-open/forwarded-tcpip-connected-port` *pkt* [Procedure]

The port to which the TCP connection was made.

`channel-open/forwarded-tcpip-originator-address` *pkt* [Procedure]

The remote address of the TCP connection.

`channel-open/forwarded-tcpip-originator-port` *pkt* [Procedure]

The remote port of the TCP connection.

`make-channel-open/session` *sender-id initial-window-size maximum-packet-size* [Procedure]

Construct a request to open a session channel. This type of channel is used for interactive logins, remote command execution, etc. After the channel has been established the client will send e.g. a `channel-request/shell` or a `channel-request/exec` request.

`channel-open/session?` *obj* [Procedure]

Returns true if *obj* is a `channel-open/session` packet.

`make-channel-open/x11` *type sender-id initial-window-size maximum-packet-size originator-address originator-port* [Procedure]

Constructs a message that opens an X11 channel. This message can be sent after X11 forwarding has been requested.

`channel-open/x11?` *obj* [Procedure]

Returns true if *obj* is a `channel-open/x11` packet.

`channel-open/x11-originator-address` *pkt* [Procedure]

The network address that originated the X11 connection.

`channel-open/x11-originator-port` *pkt* [Procedure]

The network port that originated the X11 connection.

`channel-packet?` *obj* [Procedure]

Returns true if *obj* is a `channel-packet` packet.

`channel-packet-recipient` *pkt* [Procedure]

This field is an integer that identifies the ID of the channel that should receive the request.

`make-channel-open-failure` *recipient reason-code description language* [Procedure]

Constructs a packet that represents a failure to open a channel. It is sent in response to a `channel-open/*` request.

`channel-open-failure?` *obj* [Procedure]

Returns true if *obj* is a `channel-open-failure` packet.

`channel-open-failure-reason-code` *pkt* [Procedure]

```
SSH-OPEN-ADMINISTRATIVELY-PROHIBITED
SSH-OPEN-CONNECT-FAILED
SSH-OPEN-UNKNOWN-CHANNEL-TYPE
SSH-OPEN-RESOURCE-SHORTAGE
```

`channel-open-failure-description` *pkt* [Procedure]

This field is a human-readable reason for why the channel could not be opened.

`channel-open-failure-language` *pkt* [Procedure]

This field is most commonly unused and set to `""`.

`make-channel-open-confirmation` *recipient sender initial-window-size maximum-packet-size* [Procedure]

Constructs a message that indicates a channel was successfully opened (identified by *recipient*). The party that sends this message will include its own channel ID (`sender`).

`channel-open-confirmation?` *obj* [Procedure]

Returns true if *obj* is a `channel-open-confirmation` packet.

`channel-open-confirmation-sender` *pkt* [Procedure]

This field contains the sender's ID for this channel.

`channel-open-confirmation-initial-window-size` *pkt* [Procedure]

This is the sender's initial window size. Analogous to the initial window size in a `channel-open/*` request.

`channel-open-confirmation-maximum-packet-size` *pkt* [Procedure]

This is the sender's maximum packet size. Analogous to the maximum packet size in a `channel-open/*` request.

`make-channel-window-adjust` *recipient amount* [Procedure]

This constructs a packet that is used to increment the window size of channel *recipient* by *amount* octets. It tells the remote part that the channel may receive additional data. If the client has assigned to a channel a receive buffer of 4096 bytes and the server sends 4096 bytes, the server will not be able to successfully send more data until the client has processed some of the buffer. When there is more room in the buffer the client can send a message of this type.

`channel-window-adjust?` *obj* [Procedure]

Returns true if *obj* is a `channel-window-adjust` packet.

`channel-window-adjust-amount` *pkt* [Procedure]
This field contains the number of bytes that will be added to the window size.

`make-channel-data` *recipient value* [Procedure]
This constructs a request that sends data over a channel.

`channel-data?` *obj* [Procedure]
Returns true if *obj* is a `channel-data` packet.

`channel-data-value` *pkt* [Procedure]
This field contains a bytevector with data being sent over the channel.

`make-channel-extended-data` *recipient type value* [Procedure]
This constructs a message that works just like `channel-data`, except it contains an additional *type* field (explained below).

`channel-extended-data?` *obj* [Procedure]
Returns true if *obj* is a `channel-extended-data` packet.

`channel-extended-data-type` *pkt* [Procedure]
Data sent by a `channel-data` packet will normally be sent to a port connected with standard output. A `channel-extended-data` field is used when the data destination is a different port.

`SSH-EXTENDED-DATA-STDERR`
This constant specifies that the destination is the standard error port.

`channel-extended-data-value` *pkt* [Procedure]
This field contains a bytevector with the data sent over the channel, e.g. an error message printed on the standard error port.

`make-channel-eof` *recipient* [Procedure]
This constructs a packet that signals the end-of-file condition on the channel identified by the `recipient` ID.

`channel-eof?` *obj* [Procedure]
Returns true if *obj* is a `channel-eof` packet.

`make-channel-close` *recipient* [Procedure]
This constructs a message that is used when a channel is closed.

`channel-close?` *obj* [Procedure]
Returns true if *obj* is a `channel-close` packet.

`make-channel-success` *recipient* [Procedure]
This constructs a packet that indicates that the previous request was successful. These packets are sent in response to requests where *want-reply?* is true.

`channel-success?` *obj* [Procedure]
Returns true if *obj* is a `channel-success` packet.

`make-channel-failure` *recipient* [Procedure]
This constructs a packet that indicates that the previous request was not successful. These packets are sent in response to requests where *want-reply?* is true.

`channel-failure?` *obj* [Procedure]
Returns true if *obj* is a `channel-failure` packet.

`channel-request?` *obj* [Procedure]
Returns true if *obj* is a `channel-request` packet.

`channel-request-type` *req* [Procedure]
This field is a string that identifies the type of the request, e.g. `"break"` or `"shell"`.

`channel-request-want-reply?` *req* [Procedure]
When this field is true the peer will respond with `channel-success` or `channel-failure`. This field is not valid for all requests. Where it is not valid the constructor will not include it as an argument.

`make-channel-request/break` *recipient want-reply? length* [Procedure]
This constructs a request that relays a "BREAK" signal on the channel. A "BREAK" is a signalling mechanism used with serial consoles. This request is standardized by RFC 4335.

`channel-request/break?` *obj* [Procedure]
Returns true if *obj* is a `channel-request/break` packet.

`channel-request/break-length` *req* [Procedure]
The length of the signal in milliseconds.

`make-channel-request/env` *recipient want-reply? name value* [Procedure]
Constructs a request that can be used before a shell or command has been started. It is used to set an environment variable (of the same kind that SRFI-98 accesses).

`channel-request/env?` *obj* [Procedure]
Returns true if *obj* is a `channel-request/env` packet.

`channel-request/env-name` *req* [Procedure]
This is a string that identifies the name of the environment variable.

`channel-request/env-value` *req* [Procedure]
This is a bytevector that contains the value of the environment variable.

`make-channel-request/exec` *recipient want-reply? command* [Procedure]
Constructs a request that instructs the server to execute a command. The channel identified by *recipient* will be connected to the standard input and output ports of the program started by the server.

`channel-request/exec?` *obj* [Procedure]
Returns true if *obj* is a `channel-request/exec` packet.

`channel-request/exec-command` *req* [Procedure]
This field is a bytevector that contains the command that the server should try to execute.

`make-channel-request/exit-signal` *recipient name core-dumped? message language* [Procedure]
This constructs a packet which indicates that the program connected to the channel identified by *recipient* has exited due to an operating system signal.

`channel-request/exit-signal?` *obj* [Procedure]
Returns true if *obj* is a `channel-request/exit-signal` packet.

`channel-request/exit-signal-name` *req* [Procedure]
This is a string that identifies the signal by name. For POSIX systems it is one of the following: `"ABRT"`, `"ALRM"`, `"FPE"`, `"HUP"`, `"ILL"`, `"INT"`, `"KILL"`, `"PIPE"`, `"QUIT"`, `"SEGV"`, `"TERM"`, `"USR1"`, `"USR2"`. Other signal names may be used by following the guidelines in section 6.10 of RFC 4254.

`channel-request/exit-signal-core-dumped?` *req* [Procedure]
This field is true when the operating system saved a process image ("core dump") when it sent the signal.

`channel-request/exit-signal-message` *req* [Procedure]
This may be a string that explains the signal.

`channel-request/exit-signal-language` *req* [Procedure]
This string may identify the language used in `channel-request/exit-signal-message`.

`make-channel-request/exit-status` *recipient value* [Procedure]
This constructs a packet which indicates that the program connected to the channel identified by *recipient* has exited voluntarily.

`channel-request/exit-status?` *obj* [Procedure]
Returns true if *obj* is a `channel-request/exit-status` packet.

`channel-request/exit-status-value` *req* [Procedure]
This is an integer that identifies the exit status of the program. It is the same kind of number used by the the Scheme procedure `exit`.

`make-channel-request/pty-req` *recipient want-reply? term columns rows width height modes* [Procedure]
Constructs a request that instructs the server to allocate a pseudo-terminal (PTY) for the channel identified by *recipient*. A PTY is needed for interactive programs, such as shells and Emacs.

`channel-request/pty-req?` *obj* [Procedure]
Returns true if *obj* is a `channel-request/pty-req` packet.

`channel-request/pty-req-term` *req* [Procedure]
This is a string that identifies the type of terminal that this PTY will be connected to. If the terminal is compatible with the DEC VT100 the value would be `"vt100"`. This value is also the environment variable `TERM`. The set of supported terminal types depends on the server. Typically the software running on an SSH server uses the "terminfo" database.

`channel-request/pty-req-columns` *req* [Procedure]
: This field contains the number of columns the terminal supports, e.g. `80`. The `channel-request/window-change` request can be used to update this value if the terminal supports resizing.

`channel-request/pty-req-rows` *req* [Procedure]
: This field contains the number of rows the terminal supports, e.g. `24`.

`channel-request/pty-req-width` *req* [Procedure]
: This field specifies the width of the terminal in pixels.

`channel-request/pty-req-height` *req* [Procedure]
: This field specifies the height of the terminal in pixels.

`channel-request/pty-req-modes` *req* [Procedure]
: This is a bytevector that encodes POSIX terminal modes. Unlike the size of the terminal, it is not possible to change the modes after the PTY has been created. The client should emulate a terminal set to "raw" mode and send a correct list of terminal modes. The server will then cooperate to handle the rest. This means that, unlike with telnet, the client will generally not do local "canonical" terminal processing.

`bytevector->terminal-modes` *bv* [Procedure]
: Decodes the modes from a `channel-request/pty-req`. The return value is an association list.

`terminal-modes->bytevector` *modes* [Procedure]
: The inverse of `bytevector->terminal-modes`. All modes specified by RFC 4254 can be encoded.

```
(import (industria ssh connection))
(terminal-modes->bytevector '((VINTR . 3) (VERASE . 127)))
⇒ #vu8(1 0 0 0 3 3 0 0 0 127 0)
```

`make-channel-request/shell` *recipient want-reply?* [Procedure]
: Constructs a request that starts a login shell on the channel identified by *recipient*. Normally a PTY must first have been connected to the channel.

`channel-request/shell?` *obj* [Procedure]
: Returns true if *obj* is a `channel-request/shell` packet.

`make-channel-request/signal` *recipient name* [Procedure]
: Construct a packet that sends a signal to the program connected to the channel identified by *recipient*.

`channel-request/signal?` *obj* [Procedure]
: Returns true if *obj* is a `channel-request/signal` packet.

`channel-request/signal-name` *req* [Procedure]
: This field contains a signal name of the same type as that used by `channel-request/exit-signal`.

`make-channel-request/subsystem` *recipient want-reply? name* [Procedure]
Constructs a request that a subsystem should be connected to the channel identified by *recipient*.

`channel-request/subsystem?` *obj* [Procedure]
Returns true if *obj* is a `channel-request/subsystem` packet.

`channel-request/subsystem-name` *req* [Procedure]
This field identifies the subsystem being requested, e.g. `"sftp"`.

`make-channel-request/window-change` *recipient columns rows width height* [Procedure]
Construct a message that tells the server that the terminal window associated with a channel has been resized. The channel should have a PTY (see `channel-request/pty-req`).

`channel-request/window-change?` *obj* [Procedure]
Returns true if *obj* is a `channel-request/window-change` packet.

`channel-request/window-change-columns` *req* [Procedure]
Contains the new character cell width of the terminal window.

`channel-request/window-change-rows` *req* [Procedure]
Contains the new character cell height of the terminal window.

`channel-request/window-change-width` *req* [Procedure]
Contains the new pixel width of the terminal window.

`channel-request/window-change-height` *req* [Procedure]
Contains the new pixel height of the terminal window.

`make-channel-request/x11-req` *recipient want-reply? single-connection? protocol cookie screen* [Procedure]
Constructs an X11 (X Window System) forwarding request.

`channel-request/x11-req?` *obj* [Procedure]
Returns true if *obj* is a `channel-request/x11-req` packet.

`channel-request/x11-req-single-connection?` *req* [Procedure]
If this field is true when only one X11 connection should be forwarded.

`channel-request/x11-req-protocol` *req* [Procedure]
This field identifies an X11 authentication protocol. The most common value is `"MIT-MAGIC-COOKIE-1"`.

`channel-request/x11-req-cookie` *req* [Procedure]
This is a "magic cookie" encoded as a hexadecimal string. It is used with `"MIT-MAGIC-COOKIE-1"`. It is recommended by RFC 4254 that this cookie should be different from the actual cookie used by the X11 server. When receiving a `channel-open/x11` request the cookie can be intercepted, verified and replaced with the real one.

`channel-request/x11-req-screen` *req* [Procedure]
An X11 display can have, in X jargon, multiple screens. Normally this field would be `0`.

`make-channel-request/xon-xoff` *recipient client-can-do?* [Procedure]
Constructs a message that tells the client when it can do local processing of terminal flow control (C-s and C-q).

`channel-request/xon-xoff?` *obj* [Procedure]
Returns true if *obj* is a `channel-request/xon-xoff` packet.

`channel-request/xon-xoff-client-can-do?` *req* [Procedure]
This flag is true if the client is allowed to do local processing of terminal flow control. If the flag is false then flow control is done on the server.

2.4.2 Secure Shell Transport Layer Protocol

The `(industria ssh transport)` library implements record types, parsers and formatters for the transport layer packets in SSH.

See RFC 4253 for a description of this protocol.

`register-transport` *registrar* [Procedure]
Registers the packet types for the transport layer so that they may be received and sent. A registrar may be obtained using `ssh-conn-registrar`.

`make-disconnect` *code message language* [Procedure]
Constructs a packet that closes the SSH connection. After sending or receiving this message the connection should be closed with `close-ssh`. The `ssh-error` procedure may be more convenient than manually constructing and sending a `disconnect` packet.

`disconnect?` *obj* [Procedure]
Returns `#t` if *obj* is a `disconnect` packet.

`disconnect-code` *pkt* [Procedure]
This field is an integer that represents the cause of the disconnect. The reason could be one of these (exported) constants:

```
SSH-DISCONNECT-HOST-NOT-ALLOWED-TO-CONNECT
SSH-DISCONNECT-PROTOCOL-ERROR
SSH-DISCONNECT-KEY-EXCHANGE-FAILED
SSH-DISCONNECT-RESERVED
SSH-DISCONNECT-MAC-ERROR
SSH-DISCONNECT-COMPRESSION-ERROR
SSH-DISCONNECT-SERVICE-NOT-AVAILABLE
SSH-DISCONNECT-PROTOCOL-VERSION-NOT-SUPPORTED
SSH-DISCONNECT-HOST-KEY-NOT-VERIFIABLE
SSH-DISCONNECT-CONNECTION-LOST
SSH-DISCONNECT-BY-APPLICATION
SSH-DISCONNECT-TOO-MANY-CONNECTIONS
SSH-DISCONNECT-AUTH-CANCELLED-BY-USER
SSH-DISCONNECT-NO-MORE-AUTH-METHODS-AVAILABLE
SSH-DISCONNECT-ILLEGAL-USER-NAME
```

`disconnect-message` *pkt* [Procedure]
This is a human-readable explanation for the disconnect.

`disconnect-language` *pkt* [Procedure]
Most commonly unused, `""`.

`make-ignore` *data* [Procedure]
Construct a new `ignore` packet using the bytevector *data* as the payload. These packets are ignored by receivers but can be used to make traffic analysis more difficult.

`ignore?` *obj* [Procedure]
Returns `#t` if *obj* is an `ignore` packet.

`make-unimplemented` *sequence-number* [Procedure]
This constructs a message that should be sent when a received packet type is not implemented.

`unimplemented?` *obj* [Procedure]
Returns `#t` if *obj* is an `unimplemented` packet.

`unimplemented-sequence-number` *pkt* [Procedure]
Each packet sent over an SSH connection is given an implicit sequence number. This field exactly identifies one SSH packet.

`make-debug` *always-display? message language* [Procedure]
Constructs a debug packet. It contains a message that a client or server may optionally display to the user.

`debug?` *obj* [Procedure]
Returns `#t` if *obj* is a `debug` packet.

`debug-always-display?` *pkt* [Procedure]
If this field is true then the message should be displayed.

`debug-message` *pkt* [Procedure]
This is a string containing the debugging message. If it is displayed to the user it should first be filtered.

`debug-language` *pkt* [Procedure]
Most commonly unused, `""`.

`make-service-request` *name* [Procedure]
This constructs a service request packet. The first service requested is normally `"ssh-userauth"`. See Section 2.4.3 [ssh userauth], page 42.

`service-request?` *obj* [Procedure]
Returns `#t` if *obj* is a `service-request` packet.

`service-request-name` *pkt* [Procedure]
This is the name of the service being requested, e.g. `"ssh-userauth"`.

`make-service-accept` *name* [Procedure]
Constructs a request which indicates that access to a requested service was granted.

`service-accept?` *obj* [Procedure]
Returns `#t` if *obj* is a `service-accept` packet.

`service-accept-name` *pkt* [Procedure]
This field contains the name of the service to which access was granted.

`make-kexinit` *cookie kex-algorithms server-host-key-algorithms encryption-algorithms-client-to-server encryption-algorithms-server-to-client mac-algorithms-client-to-server mac-algorithms-server-to-client compression-algorithms-client-to-server compression-algorithms-server-to-client languages-client-to-server languages-server-to-client first-kex-packet-follows? reserved* [Procedure]
Constructs a `kexinit` packet, which is used as part of the key exchange algorithm. The arguments are explained below. You probably want to use `build-kexinit-packet` instead of this procedure.

`kexinit?` *obj* [Procedure]
Returns `#t` if *obj* is a `kexinit` packet.

`kexinit-cookie` *pkt* [Procedure]
This field is a random bytevector. It is used in the key exchange to make things more difficult for an attacker.

`kexinit-kex-algorithms` *pkt* [Procedure]
A list of the supported key exchange algorithms (mostly variations on Diffie-Hellman).

`kexinit-server-host-key-algorithms` *pkt* [Procedure]
A list of the supported host key algorithms.

`kexinit-encryption-algorithms-client-to-server` *pkt* [Procedure]
A list of the supported encryption algorithms for packets sent from the client to the server.

`kexinit-encryption-algorithms-server-to-client` *pkt* [Procedure]
A list of the supported encryption algorithms for packets sent from the server to the client.

`kexinit-mac-algorithms-client-to-server` *pkt* [Procedure]
A list of the supported Message Authentication Code (MAC) algorithms for packets sent from the client to the server.

`kexinit-mac-algorithms-server-to-client` *pkt* [Procedure]
A list of the supported Message Authentication Code (MAC) algorithms for packets sent from the server to the client.

`kexinit-compression-algorithms-client-to-server` *pkt* [Procedure]
A list of the supported compression algorithms for packets sent from the client to the server. The algorithm `"none"` is currently the only implemented compression algorithm.

`kexinit-compression-algorithms-server-to-client` *pkt* [Procedure]
A list of the supported compression algorithms for packets sent from the server to the client. The algorithm `"none"` is currently the only implemented compression algorithm.

`kexinit-languages-client-to-server` *pkt* [Procedure]
Normally never used. Set to the empty list.

`kexinit-languages-server-to-client` *pkt* [Procedure]
Normally never used. Set to the empty list.

`kexinit-first-kex-packet-follows?` *pkt* [Procedure]
If this field is true then the server and client will try to cooperate in order to make the key exchange run faster over connections with high latency. This optimization only works when the server and client both prefer the same algorithms.

`kexinit-reserved` *pkt* [Procedure]
This field must be zero.

`make-newkeys` [Procedure]
Constructs a new `newkeys` packet. This message is used as part of key exchange to notify the remote side that new encryption keys are being used.

`newkeys?` *obj* [Procedure]
Returns `#t` if *obj* is a *newkeys* packet.

2.4.3 Secure Shell Authentication Protocol

The `(industria ssh userauth)` library implements record types, parsers and formatters for the authentication protocol packets in SSH.

See RFC 4252 for a more detailed description of this protocol. In this protocol the client sends packets of type `userauth-request`. The type names that start with `userauth-request/` are sub-types that contain user credentials. All other packet types documented here are sent by the server.

All user authentication requests contain a user name, a service name and a method name. The service name most commonly used is `"ssh-connection"`, which requests access to the connection protocol. See Section 2.4.1 [ssh connection], page 29.

`register-userauth` *registrar* [Procedure]
Registers the packet types for the authentication protocol so that they may be received and sent. A registrar may be obtained using `ssh-conn-registrar`.

`register-userauth-password` *registrar* [Procedure]
Registers the packet types for the password authentication protocol. This is a supplement to `register-userauth`.

`register-userauth-public-key` *registrar* [Procedure]
Registers the packet types for the public key authentication protocol. This is a supplement to `register-userauth`.

`deregister-userauth` *registrar* [Procedure]
Deregisters all authentication protocol packet types.

`make-userauth-request` *username service method* [Procedure]
Constructs a new user authentication request. This particular procedure is only good for constructing requests that use the `"none"` method. When such a request is sent to the server it will respond with a list of available authentication methods. To make a proper request use one of the `make-userauth-request/*` procedures below. Those procedures automatically include the correct *method* in the request. The *service* is normally `"ssh-connection"`. See Section 2.4.1 [ssh connection], page 29.

`userauth-request?` *obj* [Procedure]
Returns true if *obj* is a `userauth-request` packet. This includes `userauth-request/password` packets, and so on.

`userauth-request-username` *request* [Procedure]
This returns the user name field of *request*.

`userauth-request-service` *request* [Procedure]
This returns the service name field of *request*.

`userauth-request-method` *request* [Procedure]
This returns the method name field of *request*. Examples include `"none"`, `"password"` and `"publickey"`.

If the server does not like the credentials provided in a `userauth-request` it will send a `userauth-failure` packet.

`make-userauth-failure` *can-continue partial?* [Procedure]
Constructs a message that indicates to the client that the user authentication request was not successful.

`userauth-failure?` *obj* [Procedure]
Returns true if *obj* is a `userauth-failure` packet. These packets indicate the the client was denied access to the requested service. The credentials might be incorrect or the server might be requesting additional authentication requests (see below).

`userauth-failure-can-continue` *failure* [Procedure]
: This returns a list of authentication methods that "can continue", i.e. methods that might be successful given that correct credentials are provided.

`userauth-failure-partial?` *failure* [Procedure]
: This is a boolean that indicates partial success. The server might require multiple successful authentication requests (see RFC 4252).

`make-userauth-success` [Procedure]
: Constructs a packet that indicates to the client that the user authentication was successful. The client can now use the requested service (e.g. the connection protocol). This message has no fields.

`userauth-success?` *obj* [Procedure]
: Returns true if *obj* is a `userauth-success` packet.

The server can send a banner before the user authenticates. The banner might often contain a warning about unauthorized access.

`make-userauth-banner` *message language* [Procedure]
: This constructs a textual message that the server can send to the client. The client software can then display it to the user. This happens before user authentication is attempted and often contains a warning about unauthorized accesss.

`userauth-banner?` *obj* [Procedure]
: Returns true if *obj* is a `userauth-banner` packet.

`userauth-banner-message` *banner* [Procedure]
: This field is a message that the client can show to the user.

`userauth-banner-language` *banner* [Procedure]
: This field might indicate the language of the text in the banner, but is most commonly empty and not used.

The client can try to authenticate with a password. Note that the unencrypted password is seen by the server. It's important to check hosts keys to make sure you're connecting to the right server.

`make-userauth-request/password` *username service password* [Procedure]
: Constructs a user authentication request. This is a normal attempt to login with a user name and password. There is an alternative protocol for these types of login requests: the `"keyboard-interactive"` method (support is planned).

`userauth-request/password?` *obj* [Procedure]
: Returns true if *obj* is a `userauth-request/password` packet.

`userauth-request/password-value` *request* [Procedure]
: Returns the password field for this user authentication request.

The server can request that the client should change its password.

`make-userauth-password-changereq` *prompt language* [Procedure]
This constructs a password change request. Some servers might send this packet if e.g. they use a password expiry system.

`userauth-password-changereq?` *obj* [Procedure]
Returns true if *obj* is a `userauth-request/changereq` packet.

`userauth-password-changereq-prompt` *changereq* [Procedure]
This is the message to show the user when prompting for the new password.

`userauth-password-changereq-language` *changereq* [Procedure]
This is the language used in the password change request prompt.

After having received a request to change its password a client may send a `userauth-request/password-change` packet.

`make-userauth-request/password-change` *username service old new* [Procedure]
Constructs a request to authenticate the user and at the same time change the user's password. This message may be sent without having received a `userauth-request/changereq` packet. Please see section 8 of RFC 4252 for the meaning of the packet that the server will send in response to this packet.

`userauth-request/password-change?` *obj* [Procedure]
Returns true if *obj* is a `userauth-request/password-change` packet.

`userauth-request/password-change-old` *request* [Procedure]
This field contains the user's current password.

`userauth-request/password-change-new` *request* [Procedure]
This field contains the user's new password.

`make-userauth-request/public-key-query` *username service key* [Procedure]
Before performing a potentially expensive private key operation the client may ask the server if a specific key might be used to authenticate.

`userauth-request/public-key-query?` *obj* [Procedure]
Returns true if *obj* is a `userauth-request/public-key-query` packet.

`userauth-request/public-key-query-algorithm` *request* [Procedure]
This field is automatically filled in by `make-userauth-request/public-key-query` to contain the public key algorithm name of the key contained in the query.

`userauth-request/public-key-query-key` *request* [Procedure]
This field contains an SSH public key.

`make-userauth-public-key-ok` *algorithm key* [Procedure]
The server sends `userauth-public-key-ok` to indicate that the user may try to authenticate with the given key.

`userauth-public-key-ok?` *obj* [Procedure]
Returns true if *obj* is a `userauth-public-key-ok` packet.

`userauth-public-key-ok-algorithm` *request* [Procedure]
This is a copy of the algorithm name contained in the `userauth-request/public-key-query` packet.

`userauth-public-key-ok-key` *request* [Procedure]
This is a copy of the public key contained in the `userauth-request/public-key-query` packet.

`make-userauth-request/public-key` *username service public-key* [Procedure]
This procedure creates an *unsigned* request to authenticate with public key cryptography. The client may try to authenticate itself by sending a signed request to the server. The server will have a copy of the public key on file, e.g. stored in the user's `authorized_keys` file. By using the public key it can confirm that the client is possession of the corresponding private key. The packet returned by this procedure may be signed with `sign-userauth-request/public-key`.

`userauth-request/public-key?` *obj* [Procedure]
Returns true if *obj* is a `userauth-request/public-key` packet.

`userauth-request/public-key-algorithm` *request* [Procedure]
This field indicates the public key algorithm name of the public key in the request. It is automatically filled in when the request is constructed.

`userauth-request/public-key-key` *request* [Procedure]
This field contains an SSH public key object. See Section 2.4.5 [ssh public-keys], page 47.

`sign-userauth-request/public-key` *request session-id private-key* [Procedure]
This generates a signed `userauth-request/public-key` packet. It needs an unsigned *request*, which may be created with `make-userauth-request/public-key`. The *session-id* can be recovered with `ssh-conn-session-id`. The *private-key* must be a private DSA or ECDSA key (support for RSA signing is planned). The signed request uses the SSH connection's session ID and can therefore not be used with any other connection.

2.4.4 SSH private key format conversion

The `(industria ssh private-keys)` library parses SSH private keys. The formats `DSA PRIVATE KEY`, `RSA PRIVATE KEY`, `EC PRIVATE KEY` and *OPENSSH PRIVATE KEY* are supported.

Password-protected keys are not supported.

`get-ssh-private-keys` *p* [Procedure]
Read the next list of public keys from the textual input port *p*. Keys are returned either as their regular private key type (for DSA, RSA and EC keys); or as OpenSSH private key objects (see below)

Note that this returns a list. This is done to support the OpenSSH private key format, which bundles several keys (including the public keys) into a single structure.

`openssh-private-key?` *obj* [Procedure]
True if *obj* is an OpenSSH private key.

`openssh-private-key-public` *key* [Procedure]
The public key component of the OpenSSH private *key*.

`openssh-private-key-private` *key* [Procedure]
The private key component of the OpenSSH private *key*.

`openssh-private-key-comment` *key* [Procedure]
The comment on the OpenSSH private *key*. Usually the username and hostname where the key was generated.

2.4.5 SSH public key format conversion

Use `(industria ssh public-keys)` to convert public RSA, DSA, and ECDSA keys from records to the binary SSH public key format, and the other way around. SSH is the name of a network protocol for secure terminal connections defined by RFCs 4250-4254. The key format is specified by RFC 4716. ECDSA keys are specified by RFC 5656.

The types used for RSA, DSA and ECDSA keys in this library are the same types used elsewhere. The ECDSA keys must have the record type `ecdsa-sha-2-public-key`.

Future work would be to implement parsing of the various textual formats that contain Base64 public SSH keys.

`get-ssh-public-key` *p* [Procedure]
Reads a public RSA/DSA/ECDSA key encoded in the SSH public key format from the binary input port *p*.

`ssh-public-key->bytevector` *key* [Procedure]
Converts the public RSA/DSA/ECDSA *key* to the SSH public key format.

`ssh-public-key-algorithm` *key* [Procedure]
Returns the SSH algorithm identifier of *key*. For RSA keys this is `"ssh-rsa"`, for DSA keys it is `"ssh-dss"`, and for ECDSA keys it is `"ecdsa-sha2-[identifier]"` where `[identifier]` identifies the curve.

This is a legacy procedure.

`ssh-public-key-algorithm*` *key* [Procedure]
Returns a list of valid SSH algorithm identifiers for *key*. This is the same as `ssh-public-key-algorithm`, but more than one algorithm can be returned.

`ssh-public-key-fingerprint` *key* [*algorithm*] [Procedure]
The fingerprint of the RSA/DSA/ECDSA *key* in the same format used by common SSH software and specified by RFC 4716. The *algorithm* is either sha256 (default) or md5.

`ssh-public-key-random-art` *key* [*algorithm*] [Procedure]
The random art of the RSA/DSA/ECDSA key. This is a visual representation of the key can is easier for humans to distinguish than fingerprints. This is the same art that OpenSSH's VisualHostKey feature displays. The *algorithm* is either sha256 (default) or md5.

2.5 Various utilities

2.5.1 Base 64 encoding and decoding

The `(industria base64)` library provides procedures for dealing with the standard Base 64 encoding from RFC 4648 and some variations thereof. The Base 64 encoding can be used to represent arbitrary bytevectors purely in printable ASCII.

One variation of Base 64 is in the alphabet used. The standard encoding uses an alphabet that ends with `#\+` and `#\/`, but these characters are reserved in some applications. One such application is HTTP URLs, so there is a special encoding called base64url that simply uses a different alphabet.

The line length can also vary. Some applications will need Base 64 encoded strings that have no line endings at all, while other applications have 64 or 76 characters per line. For these uses the line length must be a multiple of four characters. Sometimes there is not enough input to get a multiple of four, but then the padding character `#\=` is used. Some applications don't use padding.

Some applications have their own "Base 64" encodings that encode bits in a different order. Such will be deemed magic and shall not work with this library.

`base64-encode` *bv* [*start end line-length no-padding alphabet port*] [Procedure]
Encodes the bytevector *bv* in Base 64 encoding. Optionally a range of bytes can be specified with *start* and *end*.

If a maximum line length is required, set *line-length* to an integer multiple of four (the default is `#f`). To omit padding at the end of the data, set *no-padding* or a non-false value. The *alphabet* is a string of length 64 (by default `base64-alphabet`).

The *port* is either a textual output port or `#f`, in which case this procedure returns a string.

`base64-decode` *str* [*alphabet port strict? strict-padding?*] [Procedure]
Decodes the Base 64 data in *str*. The result is written to the binary output *port* or returned as a bytevector if *port* is `#f` or omitted.

If *strict?* is true or omitted then the string has to contain pure Base 64 data and no whitespace or other extra characters. Otherwise non-alphabet characters are ignored.

If *strict-padding?* is true or omitted then the string has to be padded to a multiple of four characters.

The default alphabet is *base64-alphabet*.

`put-delimited-base64` *port type bv* [*line-length*] [Procedure]
Write the Base 64 encoding of *bv* to the *port*. The output is delimited by BEGIN/END lines that include the *type*.

```
(import (industria base64))
(put-delimited-base64 (current-output-port) "EXAMPLE"
                      (string->utf8 "POKEY THE PENGUIN"))
⊣ -----BEGIN EXAMPLE-----
⊣ UE9LRVkgVEhFIFBFTkdVSU4=
⊣ -----END EXAMPLE-----
```

`get-delimited-base64` *port* [*strict?*] [Procedure]
Reads a delimited Base 64 encoded bytevector and returns two values: *type* (a string) and *data* (a bytevector). The *data* value is the end-of-file object if `port-eof?` would return `#t`.

Note: This procedure ignores MIME headers. Some delimited Base 64 formats have headers on the line after BEGIN, followed by an empty line.

Note: This procedure ignores the Radix-64 checksum. The Radix-64 format (RFC 4880) is based on Base 64, but appends a CRC-24 (prefixed by `#\=`) at the end of the data.

The rationale for ignoring headers and checksums is that it follows the Principle of Robustness: "Be conservative in what you send; be liberal in what you accept from others." Lines before the BEGIN line are also ignored, because some applications (like OpenSSL) like to prepend a human readable version of the data.

You should probably use special parsers if you are reading data with headers or checksums. For some applications, e.g. MIME, you would also set *strict?* to `#f`.

```
(get-delimited-base64
 (open-string-input-port
  "-----BEGIN EXAMPLE-----\n\
AAECAwQFBg==\n\
-----END EXAMPLE-----\n"))
⇒ "EXAMPLE"
⇒ #vu8(0 1 2 3 4 5 6)
```

`base64-alphabet` [Constant]
The alphabet used by the standard Base 64 encoding. The alphabet is `#\A`–`#\Z`, `#\a`–`#\z`, `#\0`–`#\9`, `#\+`, `#\/`.

`base64url-alphabet` [Constant]
The alphabet used by the base64url encoding. The alphabet is `#\A`–`#\Z`, `#\a`–`#\z`, `#\0`–`#\9`, `#\-`, `#\_`.

Version history:

- `Industria 1.5` – The decoder was optimized and the *strict?* argument was introduced.

2.5.2 Bit-string data type

The `(industria bit-strings)` library provides a data type for representing strings of bits.

`make-bit-string` *length bytevector* [Procedure]
Returns a new bit-string of *length* bits, which are aligned to the start of the *bytevector* (with zero padding bits at the end).

`bit-string-unused` *bit-string* [Procedure]
Return the number of unused bits at the end of the bytevector representation of *bit-string*.

`bit-string->integer` *bit-string* [Procedure]
Return the integer representation of *bit-string* (a non-negative exact integer).

`bit-string-bit-set?` *bit-string idx* [Procedure]
True if bit *idx* of *bit-string* is set.

`bytevector->bit-string` *bytevector length* [Procedure]
Same as `make-bit-string`.

`integer->bit-string` *int length* [Procedure]
Return a new bit-string of *length* bits which represents the integer *int*.

`bit-string=?` *bit-string0 bit-string1* [Procedure]
True if *bit-string0* equals *bit-string1* (same length and bit values) and false otherwise.

2.5.3 Bytevector utilities

The `(industria bytevectors)` library contains utilities for working with R6RS bytevectors.

`bytevector-append` [*bytevector ...*] [Procedure]
Appends the given bytevectors.

`bytevector-concatenate` *list* [Procedure]
list is a list of bytevectors. The bytevectors are appended.

`subbytevector` *bytevector start* [*end*] [Procedure]
Analogous to `substring`. Returns a new bytevector containing the bytes of *bytevector* from index `start` to *end* (exclusive).

`bytevector-for-each` *proc bytevector* [Procedure]
Apply *proc* to each byte in *bytevector*, in left-to-right order.

`bytevector-u8-index` *bytevector byte* [*start end*] [Procedure]
Searches *bytevector* for *byte*, from left to right. The optional arguments *start* and *end* give the range to search. By default the whole bytevector is searched. Returns `#f` is no match is found.

`bytevector-u8-index-right` *bytevector byte* [*start end*] [Procedure]
Analogous to `bytevector-u8-index-right`, except this procedure searches right-to-left.

`bytevector->uint` *bytevector* [*endian*] [Procedure]
bytevector is interpreted as an unsigned integer in (by default) big endian byte order and is converted to an integer. The empty bytevector is treated as zero.

`bytevector->sint` *bytevector* [*endian*] [Procedure]
bytevector is interpreted as a signed integer in (by default) big endian byte order and is converted to an integer. The empty bytevector is treated as zero.

`uint->bytevector` *integer* [*endian length*] [Procedure]
integer is converted to an unsigned integer in (by default) big endian byte order. The returned bytevector has the minimum possible length, unless *length* is specified. Zero is converted to the empty bytevector.

```
(import (industria bytevectors))
```

```
(uint->bytevector 256)
⇒ #vu8(1 0)
(uint->bytevector 255)
⇒ #vu8(255)
```

`sint->bytevector` *integer* [*endian length*] [Procedure]
integer is converted to an signed integer in (by default) big endian byte order. The returned bytevector has the minimum possible length, unless *length* is specified (it needs one more bit than `uint->bytevector`). Zero is converted to the empty bytevector.

`bytevector=?/constant-time` *bytevector1 bytevector2* [Procedure]
True if *bytevector1* and *bytevector2* are of equal length and have the same contents.

This is a drop-in replacement for `bytevector=?` that does not leak information about the outcome of the comparison by how much time the comparison takes to perform. It works by accumulating the differences between the bytevectors. This kind of operation is most often needed when comparing fixed-length message digests, so the length comparison is done in the obvious (fast) way.

2.5.4 Password hashing

The procedure provided by `(industria crypto password)` is the same type of procedure that is called `crypt` in the standard C library. It is used for password hashing, i.e. it scrambles passwords. This is a method often used when passwords need to be stored in databases.

The scrambling algorithms are based on cryptographic primitives but have been modified so that they take more time to compute. They also happen to be quite annoying to implement.

Only DES and MD5 based hashes are currently supported.

`crypt` *password salt* [Procedure]
Scrambles a *password* using the given *salt*. The *salt* can also be a hash. The returned hash will be prefixed by the salt.

A fresh random salt should be used when hashing a new password. The purpose of the salt is to make it infeasible to reverse the hash using lookup tables.

To verify that a password matches a hash, you can do something like `(string=? hash (crypt password hash))`.

```
(import (industria crypto password))
(crypt "test" "..")
⇒ "..9sjyf8zL76k"

(crypt "test" "$1$RQ3YWMJd$")
⇒ "$1$RQ3YWMJd$oIomUD5DCxenAs2icezcn."

(string=? "$1$ggKHY.Dz$fNBcmNFTa1BFGXoLsRDkS."
          (crypt "test" "$1$ggKHY.Dz$fNBcmNFTa1BFGXoLsRDkS."))
⇒ #t
```

2.5.5 Basic TCP client connections

The `(industria tcp)` provides a simple TCP client. This library needs implementation-specific code, so the author is not eager to provide more than the bare minimum.

This library should work with Ikarus Scheme, GNU Guile, Larceny (not tested with Petit Larceny and Common Larceny), Mosh Scheme, Petite Chez Scheme (as long as the nc command is installed), Vicare Scheme, and Ypsilon Scheme. Once upon a time it also worked with PLT Scheme, but it has not been tested with Racket.

Some newer alternatives to this library are SRFI-106 and `https://github.com/ktakashi/r6rs-usocket`.

`tcp-connect` *hostname portname* [Procedure]

Initiates a TCP connection to the given *hostname* and *portname* (both of which are strings).

Returns an input-port and an output-port. They are not guaranteed to be distinct.

3 Demo programs

The programs directory contains small demonstration of the libraries. These scripts are implemented in the way recommended by R6RS non-normative appendix D.

If you're packaging these libraries then I would recommend against installing the demos in the default program search path.

3.1 checksig – verifies OpenPGP signature files

This program takes a detached ascii armored OpenPGP signature, a file to check against, and a GPG keyring. It then verifies the signature. As a curiosity it also prints OpenSSH-style random art for the key that made the signature.

3.2 honingsburk – simple Secure Shell honey pot

This demonstrates the server part of the SSH library. It starts up a dummy SSH server that accepts logins with the username root and the password toor. The server does not create a real PTY and the client does not gain access to the computer running the server. It presents a command line where all commands return an error. It uses a few non-standard procedures from Ikarus.

3.3 secsh-client – manually operated Secure Shell client

Most SSH clients try to provide a nice user experience. This one is instead a command-line based manually operated client. After establishing the initial connection you can use a few simplistic commands to login, establish a session channel, read and write channel data. You can also enable debugging if you'd like to see a packet trace. This session log shows how to connect to a `honingsburk` running on TCP port 2222:

```
Industria SSH demo client.

Connecting to localhost port 2222...
Running key exchange...
a6:4b:7e:05:38:03:01:29:07:0c:58:a4:fe:c1:d8:02
+---[ECDSA 521]---+
|*++o..           |
|ooo .            |
|Eo   . .         |
|o +   + .        |
| + +   oS.       |
|  o .  o  .      |
|   .  o  .       |
|     o ..        |
|      o.         |
+-----------------+

localhost ecdsa-sha2-nistp521 AAAAE2VjZHNhLXNoYTItbmlzdHA1[...]
Please verify the above key.
```

```
SSH session established.
Type help for a list of commands.

localhost=> u "root"
Your request to use ssh-userauth was accepted.
You may try these authentication methods: (password)
localhost=> p "toor"
You've succesfully authenticated.
You now have access to the SSH connection protocol.
localhost=> s
New session opened.
Receive side parameters:
ID: 0 window size: 4096 maximum packet size: 32768
Send side parameters:
ID: 0 window size: 32768 maximum packet size: 32768
localhost=> t 0
localhost=> r
Linux darkstar 2.6.35.8 #1 Sat Oct 30 10:43:19 CEST 2010 i686

Welcome to your new account!
No mail.
localhost=> r

darkstar:~#
localhost=>
```

Index

D

E

F

G

H

I

K

M

N

O

P

R

S

T

U

V

X