Crypto Application

version 1.1

Typeset in LaTeX from SGML source using the DOCBUILDER 3.2.2 Document System.

Contents

Chapter 1

Crypto Release Notes

The Crypto Application provides functions for computation of message digests, and encryption and decryption functions.

1.1 Crypto Release Notes

This document describes the changes made to the Crypto application.

1.1.1 Crypto 1.1.2

Reported Fixed Bugs and Malfunctions

- In the manual page `crypto(3)` the function names `md5_finish` and `sha_finish` have been changed to `md5_final` and `sha_final` to correctly document the implementation. Own Id: OTP-3409

1.1.2 Crypto 1.1.1

Code replacement in runtime is supported. Upgrade can be done from from version 1.1 and downgrade to version 1.1.

Improvements and New Features

- The driver part of the Crypto application has been updated to use the erl_driver header file. Version 1.1.1 requires emulator version 4.9.1 or later.

1.1.3 Crypto 1.1

Reported Fixed Bugs and Malfunctions

- On Windows the crypto_drv was incorrectly linked to static run-time libraries instead of dynamic ones. Own Id: OTP-3240

1.1.4 Crypto 1.0

New application.

Crypto Reference Manual

Short Summaries

- Application crypto [page 5] – The Crypto Application
- Erlang Module crypto [page 6] – Crypto Functions

crypto

No functions are exported.

crypto

The following functions are exported:

- `start() -> ok`
  [page 6] Start the crypto server.
- `stop() -> ok`
  [page 6] Stop the crypto server.
- `info() -> [atom()]`
  [page 6] Provide a list of available crypto functions.
- `md5(Data) -> Digest`
  [page 6] Compute an `MD5` message digest from `Data`
- `md5_init() -> Context`
  [page 6] Creates an MD5 context
- `md5_update(Context, Data) -> NewContext`
  [page 7] Update an MD5 `Context` with `Data`, and return a `NewContext`
- `md5_final(Context) -> Digest`
  [page 7] Finish the update of an MD5 `Context` and return the computed `MD5` message digest
- `sha(Data) -> Digest`
  [page 7] Compute an `SHA` message digest from `Data`
- `sha_init() -> Context`
  [page 7] Create an SHA context
- `sha_update(Context, Data) -> NewContext`
  [page 7] Update an SHA context
- `sha_final(Context) -> Digest`
  [page 7] Finish the update of an SHA context

- `md5_mac(Key, Data) -> Mac`
  [page 7] Compute an MD5 MAC message authentification code
- `md5_mac_96(Key, Data) -> Mac`
  [page 8] Compute an MD5 MAC message authentification code
- `sha_mac(Key, Data) -> Mac`
  [page 8] Compute an MD5 MAC message authentification code
- `sha_mac_96(Key, Data) -> Mac`
  [page 8] Compute an MD5 MAC message authentification code
- `des_cbc_encrypt(Key, IVec, Text) -> Cipher`
  [page 8] Encrypt `Text` according to DES in CBC mode
- `des_cbc_decrypt(Key, IVec, Cipher) -> Text`
  [page 8] Decrypt `Cipher` according to DES in CBC mode

crypto

Application

This chapter describes the application Crypto in OTP, which provides message digests MD5 and SHA, and CBC-DES encryption and decryption.

The purpose of this application is to provide message digest and DES encryption for SMNPv3.

Configuration

The following environment configuration parameters are defined for the Crypto application. Refer to application(3) for more information about configuration parameters.

`debug = true | false <optional>` Causes debug information to be written to standard error or standard output. Default is `false`.

SEE ALSO

application(3)

crypto

Erlang Module

This module provides a set of cryptographic functions.

References:

- md5: The MD5 Message Digest Algorithm (RFC 1321)
- sha: Secure Hash Standard (FIPS 180-1)
- hmac: Keyed-Hashing for Message Authentication (RFC 2104)
- des: Data Encryption Standard (FIPS 46-2)
- ecb, cbc, cfb, ofb: DES modes of operation (FIPS 81).

Types

```
byte() = 0 ... 255
ioelem() = byte() | binary() | iolist()
iolist() = [ioelem()]
```

Exports

`start() -> ok`

Starts the crypto server.

`stop() -> ok`

Stops the crypto server.

`info() -> [atom()]`

Stops the crypto server.

`md5(Data) -> Digest`

Types:

- Data = iolist() | binary()
- Digest = binary()

Computes an `MD5` message digest from `Data`, where the length of the digest is 128 bits (16 bytes).

`md5_init() -> Context`

Types:

- Context = binary()

Creates an MD5 context, to be used in subsequent calls to `md5_update/2`.

`md5_update(Context, Data) -> NewContext`

Types:

- Data = iolist() | binary()
- Context = NewContext = binary()

Updates an MD5 `Context` with `Data`, and returns a `NewContext`.

`md5_final(Context) -> Digest`

Types:

- Context = Digest = binary()

Finishes the update of an MD5 `Context` and returns the computed `MD5` message digest.

`sha(Data) -> Digest`

Types:

- Data = iolist() | binary()
- Digest = binary()

Computes an `SHA` message digest from `Data`, where the length of the digest is 160 bits (20 bytes).

`sha_init() -> Context`

Types:

- Context = binary()

Creates an SHA context, to be used in subsequent calls to `sha_update/2`.

`sha_update(Context, Data) -> NewContext`

Types:

- Data = iolist() | binary()
- Context = NewContext = binary()

Updates an SHA `Context` with `Data`, and returns a `NewContext`.

`sha_final(Context) -> Digest`

Types:

- Context = Digest = binary()

Finishes the update of an SHA `Context` and returns the computed `SHA` message digest.

`md5_mac(Key, Data) -> Mac`

Types:

- Key = Data = iolist() | binary()
- Mac = binary()

Computes an MD5 MAC message authentification code from Key and Data, where the the length of the Mac is 128 bits (16 bytes).

`md5_mac_96(Key, Data) -> Mac`

Types:

- Key = Data = iolist() | binary()
- Mac = binary()

Computes an MD5 MAC message authentification code from Key and Data, where the length of the Mac is 96 bits (12 bytes).

`sha_mac(Key, Data) -> Mac`

Types:

- Key = Data = iolist() | binary()
- Mac = binary()

Computes an SHA MAC message authentification code from Key and Data, where the length of the Mac is 160 bits (20 bytes).

`sha_mac_96(Key, Data) -> Mac`

Types:

- Key = Data = iolist() | binary()
- Mac = binary()

Computes an SHA MAC message authentification code from Key and Data, where the length of the Mac is 96 bits (12 bytes).

`des_cbc_encrypt(Key, IVec, Text) -> Cipher`

Types:

- Key = Text = iolist() | binary()
- IVec = Cipher = binary()

Encrypts Text according to DES in CBC mode. Text must be a multiple of 64 bits (8 bytes). Key is the DES key, and IVec is an arbitrary initializing vector. The lengths of Key and IVec must be 64 bits (8 bytes).

`des_cbc_decrypt(Key, IVec, Cipher) -> Text`

Types:

- Key = Cipher = iolist() | binary()
- IVec = Text = binary()

Decrypts Cipher according to DES in CBC mode. Key is the DES key, and IVec is an arbitrary initializing vector. Key and IVec must have the same values as those used when encrypting. Cipher must be a multiple of 64 bits (8 bytes). The lengths of Key and IVec must be 64 bits (8 bytes).

DES in CBC mode

The Data Encryption Standard (DES) defines an algoritm for encrypting and decrypting an 8 byte quantity using an 8 byte key (actually only 56 bits of the key is used).

When it comes to encrypting and decrypting blocks that are multiples of 8 bytes various modes are defined (FIPS 81). One of those modes is the Cipher Block Chaining (CBC) mode, where the encryption of an 8 byte segment depend not only of the contents of the segment itself, but also on the result of encrypting the previous segment: the encryption of the previous segment becomes the initializing vector of the encryption of the current segment.

Thus the encryption of every segment depends on the encryption key (which is secret) and the encryption of the previous segment, except the first segment which has to be provided with a first initializing vector. That vector could be chosen at random, or be counter of some kind. It does not have to be secret.

The following example is drawn from the FIPS 81 standard, where both the plain text and the resulting cipher text is settled. We use the Erlang bitsyntax to define binary literals. The following Erlang code fragment returns 'true'.

```
Key = <<16#01,16#23,16#45,16#67,16#89,16#ab,16#cd,16#ef>>,
IVec = <<16#12,16#34,16#56,16#78,16#90,16#ab,16#cd,16#ef>>,
P = "Now is the time for all ",
C = crypto:des_cbc_encrypt(K, I, P),
C == <<16#e5,16#c7,16#cd,16#de,16#87,16#2b,16#f2,16#7c,
       16#43,16#e9,16#34,16#00,16#8c,16#38,16#9c,16#0f,
       16#68,16#37,16#88,16#49,16#9a,16#7c,16#05,16#f6>>,
<<"Now is the time for all ">> ==
                 crypto:des_cbc_decrypt(Key,IVec,C).
```

The following is true for the DES CBC mode. For all decompositions `P1 ++ P2 = P` of a plain text message `P` (where the length of all quantities are multiples of 8 bytes), the encryption `C` of `P` is equal to `C1 ++ C2`, where `C1` is obtained by encrypting `P1` with `Key` and the initializing vector `IVec`, and where `C2` is obtained by encrypting `P2` with `Key` and the initializing vector `l(C1)`, where `l(B)` denotes the last 8 bytes of the binary `B`.

Similarly, for all decompositions `C1 ++ C2 = C` of a cipher text message `C` (where the length of all quantities are multiples of 8 bytes), the decryption `P` of `C` is equal to `P1 ++ P2`, where `P1` is obtained by decrypting `C1` with `Key` and the initializing vector `IVec`, and where `P2` is obtained by decrypting `C2` with `Key` and the initializing vector `l(C1)`, where `l(.)` is as above.

Index of Modules and Functions

Modules are typed in *this way*.
Functions are typed in `this way`.