

Megaco/H.248

Copyright © 2000-2025 Ericsson AB. All Rights Reserved.

Megaco/H.248 3.18.7

October 21, 2025

Copyright © 2000-2025 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

October 21, 2025

1 Megaco/H.248 Users Guide

The Megaco application is a framework for building applications on top of the Megaco/H.248 protocol.

1.1 Introduction

Megaco/H.248 is a protocol for control of elements in a physically decomposed multimedia gateway, enabling separation of call control from media conversion. A Media Gateway Controller (MGC) controls one or more Media Gateways (MG).

This version of the stack supports version 1, 2 and 3 as defined by:

- version 1 - RFC 3525 and H.248-IG (v10-v13)
- version 2 - draft-ietf-megaco-h248v2-04 & H.248.1 v2 Corrigendum 1 (03/2004)
- version 3:
  - prev3a - as defined by TD-33 (except segments)
  - prev3b - TD-33 updated to be backward compatible with v2 (except segments)
  - prev3c - As defined by ITU H.248.1 (09/2005) (except segments)
  - v3 - Full version 3 as defined by ITU H.248.1 (09/2005) (including segments)

The semantics of the protocol has jointly been defined by two standardization bodies:

- IETF - which calls the protocol Megaco
- ITU - which calls the protocol H.248

1.1.1 Scope and Purpose

This manual describes the Megaco application, as a component of the Erlang/Open Telecom Platform development environment. It is assumed that the reader is familiar with the Erlang Development Environment, which is described in a separate User's Guide.

1.1.2 Prerequisites

The following prerequisites is required for understanding the material in the Megaco User's Guide:

- the basics of the Megaco/H.248 protocol
- the basics of the Abstract Syntax Notation One (ASN.1)
- familiarity with the Erlang system and Erlang programming

The application requires Erlang/OTP release R10B or later.

1.1.3 About This Manual

In addition to this introductory chapter, the Megaco User's Guide contains the following chapters:

- Chapter 2: "Architecture" describes the architecture and typical usage of the application.
- Chapter 3: "Internal form and its encodings" describes the internal form of Megaco/H.248 messages and its various encodings.
- Chapter 4: "Transport mechanisms" describes how different mechanisms can be used to transport the Megaco/H.248 messages.

- Chapter 5: "Debugging" describes tracing and debugging.

1.1.4 Where to Find More Information

Refer to the following documentation for more information about Megaco/H.248 and about the Erlang/OTP development system:

- **version 1, RFC 3525**
- **old version 1, RFC 3015**
- **Version 2 Corrigendum 1**
- **version 2, draft-ietf-megaco-h248v2-04**
- **TD-33 (Draft H.248.1 version 3)**
- **H.248.1 version 3**
- the ASN.1 application User's Guide
- the Megaco application Reference Manual
- Concurrent Programming in Erlang, 2nd Edition (1996), Prentice-Hall, ISBN 0-13-508301-X.

1.2 Architecture

1.2.1 Network view

Megaco is a (master/slave) protocol for control of gateway functions at the edge of the packet network. Examples of this is IP-PSTN trunking gateways and analog line gateways. The main function of Megaco is to allow gateway decomposition into a call agent (call control) part (known as Media Gateway Controller, MGC) - master, and an gateway interface part (known as Media Gateway, MG) - slave. The MG has no call control knowledge and only handle making the connections and simple configurations.

SIP and H.323 are peer-to-peer protocols for call control (valid only for some of the protocols within H.323), or more generally multi-media session protocols. They both operate at a different level (call control) from Megaco in a decomposed network, and are therefor not aware of whether or not Megaco is being used underneath.

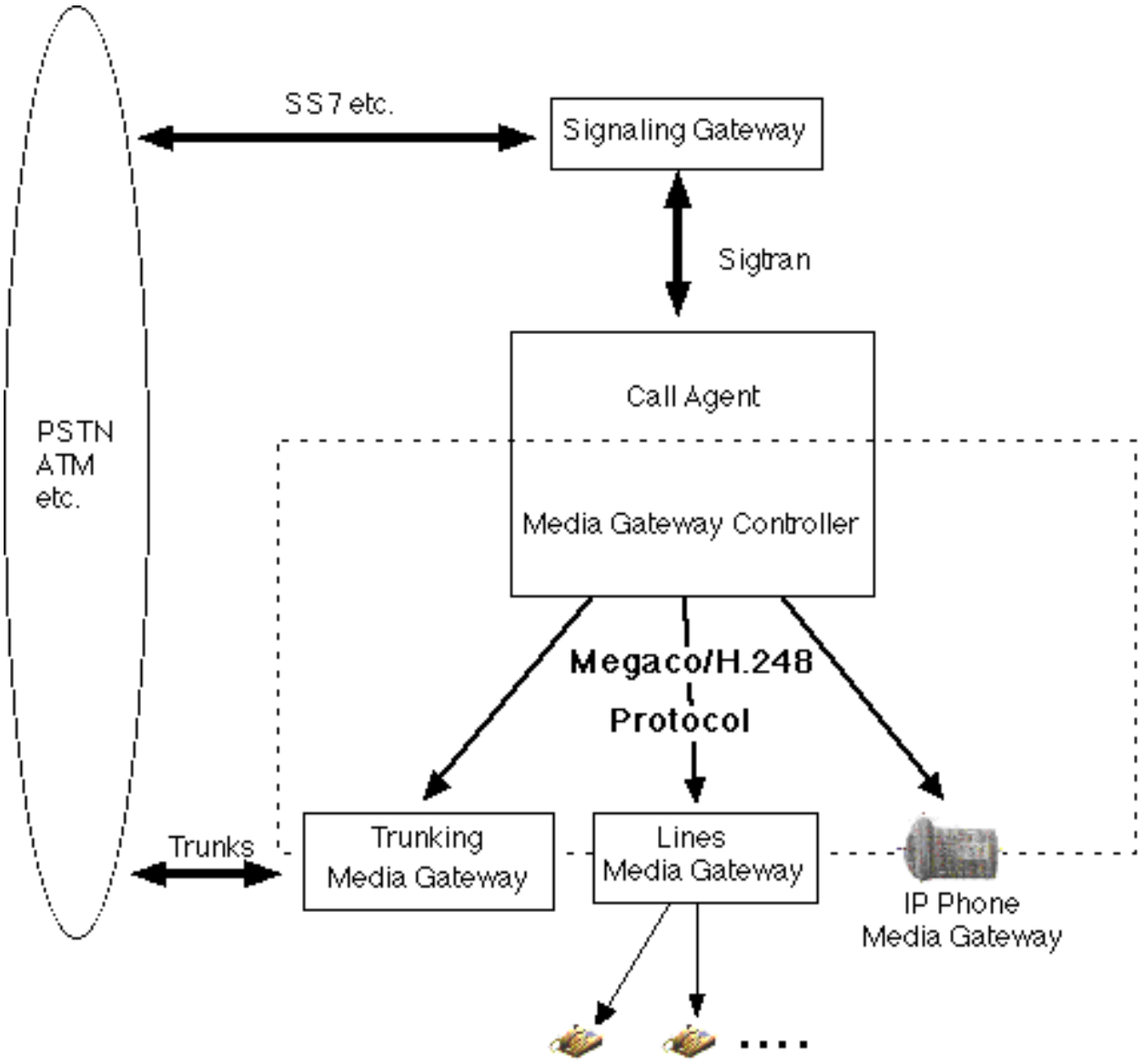


Figure 2.1: Network architecture

Megaco and peer protocols are complementary in nature and entirely compatible within the same system. At a system level, Megaco allows for

- overall network cost and performance optimization
- protection of investment by isolation of changes at the call control layer
- freedom to geographically distribute both call function and gateway function
- adaption of legacy equipment

1.2.2 General

This Erlang/OTP application supplies a framework for building applications that needs to utilize the Megaco/H.248 protocol.

We have introduced the term "user" as a generic term for either an MG or an MGC, since most of the functionality we support, is common for both MG's and MGC's. A (local) user may be configured in various ways and it may establish any number of connections to its counterpart, the remote user. Once a connection has been established, the connection is supervised and it may be used for the purpose of sending messages. N.B. according to the standard an MG is connected to at most one MGC, while an MGC may be connected to any number of MG's.

For the purpose of managing "virtual MG's", one Erlang node may host any number of MG's. In fact it may host a mix of MG's and MGC's. You may say that an Erlang node may host any number of "users".

The protocol engine uses callback modules to handle various things:

- encoding callback modules - handles the encoding and decoding of messages. Several modules for handling different encodings are included, such as ASN.1 BER, pretty well indented text, compact text and some others. Others may be written by you.
- transport callback modules - handles sending and receiving of messages. Transport modules for TCP/IP and UDP/IP are included and others may be written by you.
- user callback modules - the actual implementation of an MG or MGC. Most of the functions are intended for handling of a decoded transaction (request, reply, acknowledgement), but there are others that handles connect, disconnect and errors cases.

Each connection may have its own configuration of callback modules, re-send timers, transaction id ranges etc. and they may be re-configured on-the-fly.

In the API of Megaco, a user may explicitly send action requests, but generation of transaction identifiers, the encoding and actual transport of the message to the remote user is handled automatically by the protocol engine according to the actual connection configuration. Megaco messages are not exposed in the API.

On the receiving side the transport module receives the message and forwards it to the protocol engine, which decodes it and invokes user callback functions for each transaction. When a user has handled its action requests, it simply returns a list of action replies (or a message error) and the protocol engine uses the encoding module and transport module to compose and forward the message to the originating user.

The protocol stack does also handle things like automatic sending of acknowledgements, pending transactions, re-send of messages, supervision of connections etc.

In order to provide a solution for scalable implementations of MG's and MGC's, a user may be distributed over several Erlang nodes. One of the Erlang nodes is connected to the physical network interface, but messages may be sent from other nodes and the replies are automatically forwarded back to the originating node.

1.2.3 Single node config

Here a system configuration with an MG and MGC residing in one Erlang node each is outlined:

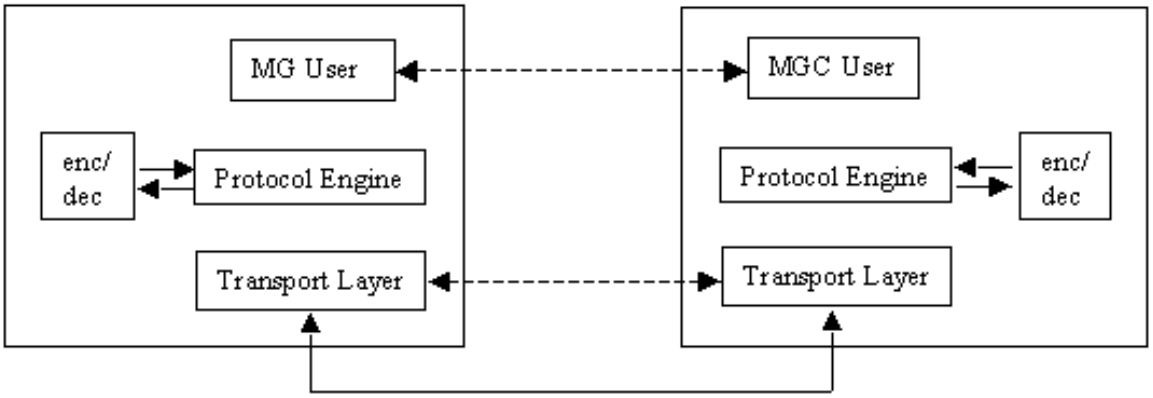


Figure 2.2: Single node config

1.2.4 Distributed config

In a larger system with a user (in this case an MGC) distributed over several Erlang nodes, it looks a little bit different. Here the encoding is performed on the originating Erlang node (1) and the binary is forwarded to the node (2) with the physical network interface. When the potential message reply is received on the interface on node (2), it is decoded there and then different actions will be taken for each transaction in the message. The transaction reply will be forwarded in its decoded form to the originating node (1) while the other types of transactions will be handled locally on node (2).

Timers and re-send of messages will be handled on locally on one node, that is node(1), in order to avoid unnecessary transfer of data between the Erlang nodes.

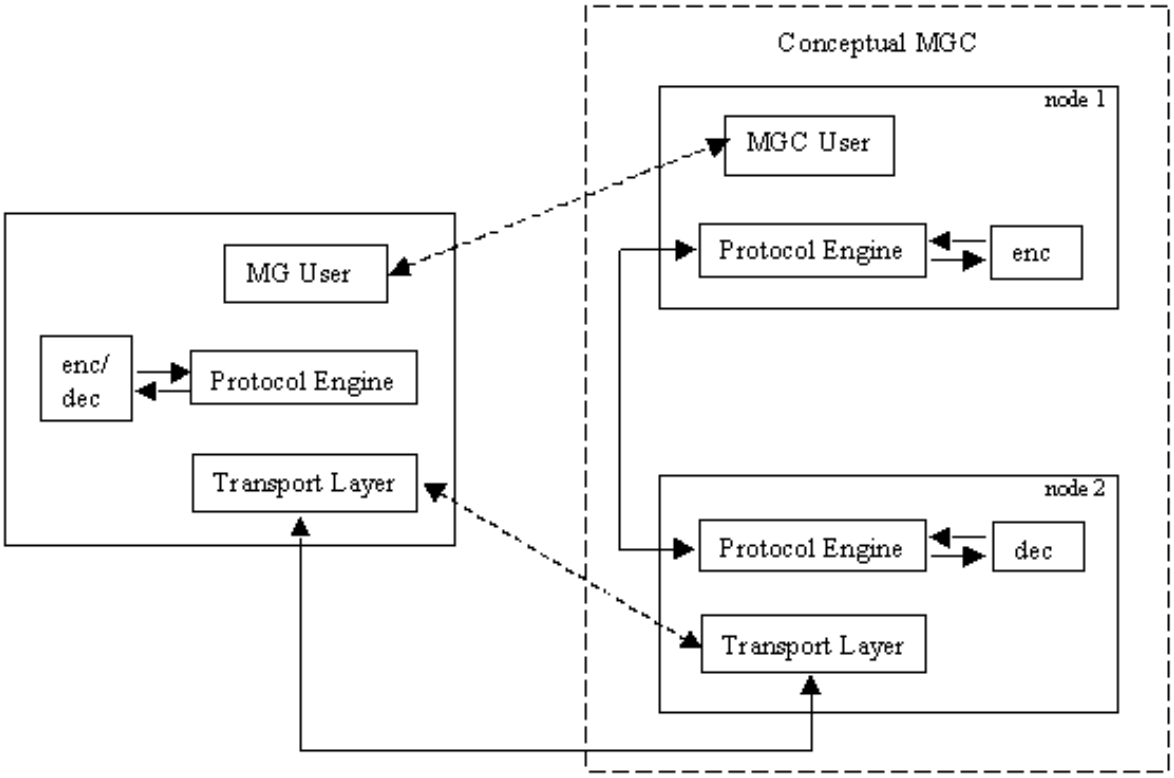


Figure 2.3: Distributes node config

1.2.5 Message round-trip call flow

The typical round-trip of a message can be viewed as follows. Firstly we view the call flow on the originating side:

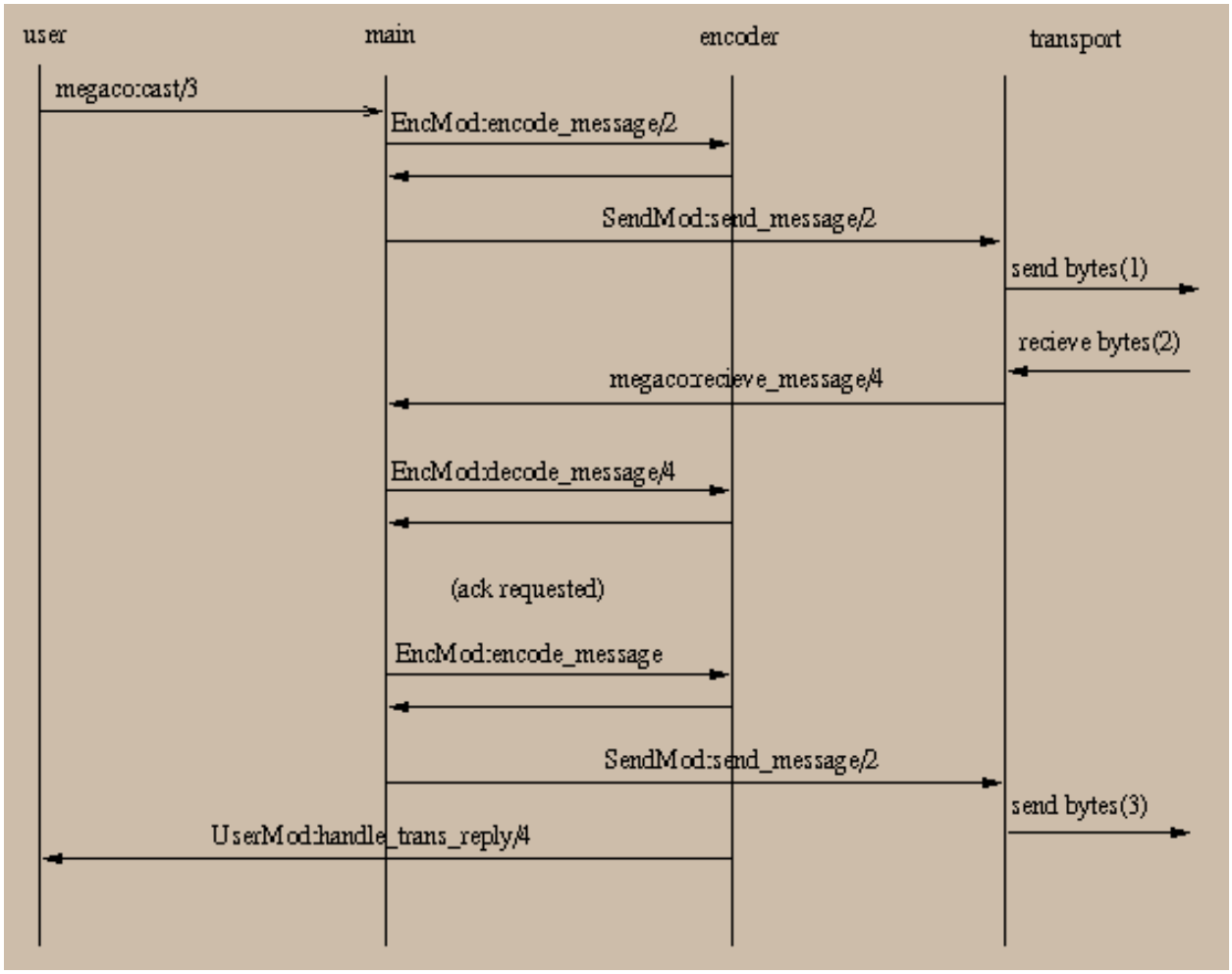


Figure 2.4: Message Call Flow (originating side)

Then we continue with the call flow on the destination side:

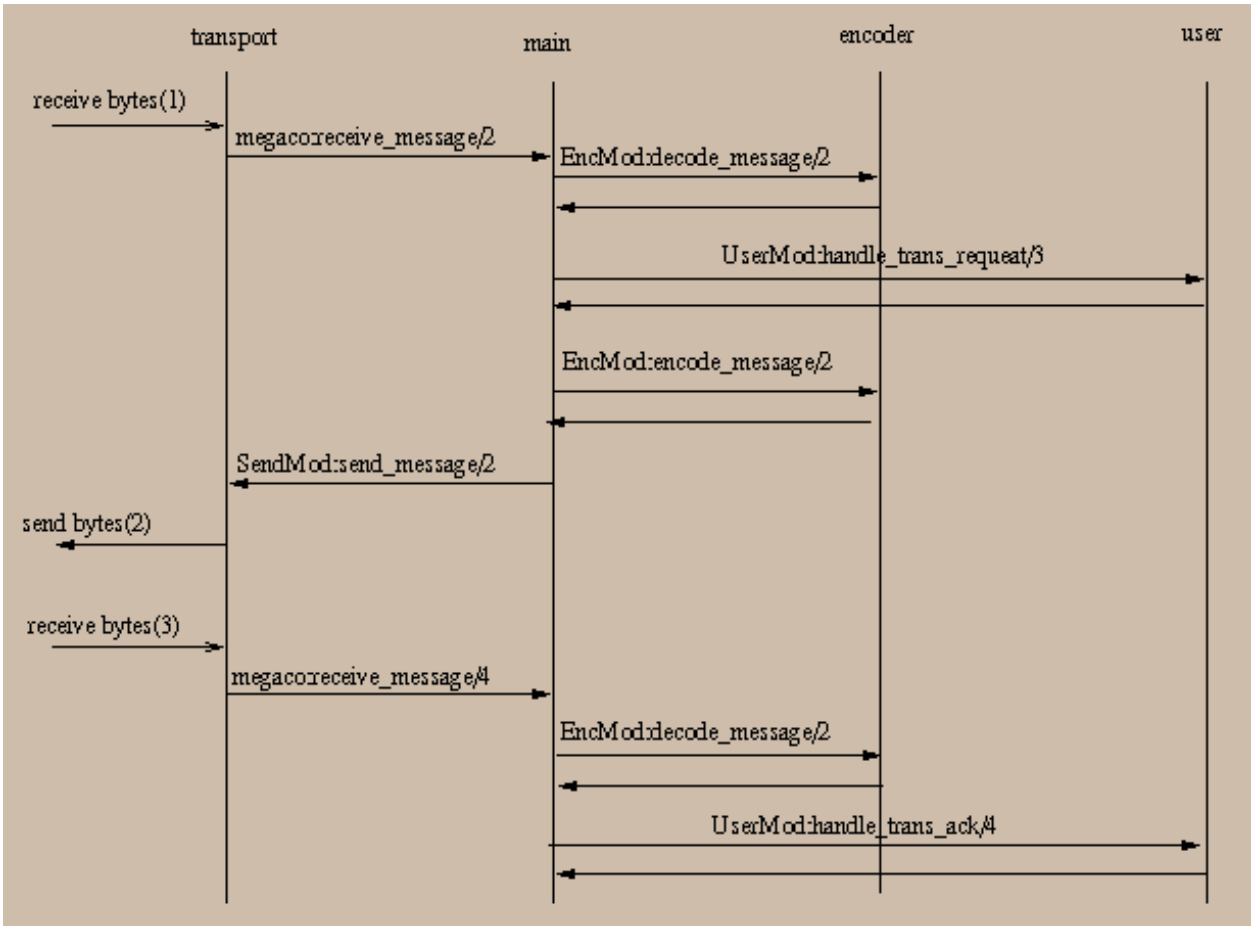


Figure 2.5: Message Call Flow (destination side)

1.3 Running the stack

1.3.1 Starting

A user may have a number of "virtual" connections to other users. An MG is connected to at most one MGC, while an MGC may be connected to any number of MG's. For each connection the user selects a transport service, an encoding scheme and a user callback module.

An MGC must initiate its transport service in order to listen to MG's trying to connect. How the actual transport is initiated is outside the scope of this application. However a send handle (typically a socket id or host and port) must be provided from the transport service in order to enable us to send the message to the correct destination. We do however not assume anything about this, from our point of view, opaque handle. Hopefully it is rather small since it will passed around the system between processes rather frequently.

A user may either be statically configured in a .config file according to the application concept of Erlang/OTP or dynamically started with the configuration settings as arguments to megaco:start_user/2. These configuration settings may be updated later on with megaco:update_conn_info/2.

The function megaco:connect/4 is used to tell the Megaco application about which control process it should supervise, which MID the remote user has, which callback module it should use to send messages etc. When this "virtual" connection is established the user may use megaco:call/3 and megaco:cast/3 in order to send messages to the other

side. Then it is up to the MG to send its first Service Change Request message after applying some clever algorithm in order to fight the problem with startup avalanche (as discussed in the RFC).

The originating user will wait for a reply or a timeout (defined by the request_timer). When it receives the reply this will optionally be acknowledged (regulated by auto_ack), and forwarded to the user. If an interim pending reply is received, the long_request_timer will be used instead of the usual request_timer, in order to enable avoidance of spurious re-sends of the request.

On the destination side the transport service waits for messages. Each message is forwarded to the Megaco application via the megaco:receive_message/4 callback function. The transport service may or may not provide means for blocking and unblocking the reception of the incoming messages.

If a message is received before the "virtual" connection has been established, the connection will be setup automatically. An MGC may be real open minded and dynamically decide which encoding and transport service to use depending on how the transport layer contact is performed. For IP transports two ports are standardized, one for textual encoding and one for binary encoding. If for example an UDP packet was received on the text port it would be possible to decide encoding and transport on the fly.

After decoding a message various user callback functions are invoked in order to allow the user to act properly. See the megaco_user module for more info about the callback arguments.

When the user has processed a transaction request in its callback function, the Megaco application assembles a transaction reply, encodes it using the selected encoding module and sends the message back by invoking the callback function:

- SendMod:send_message(SendHandle, ErlangBinary)

Re-send of messages, handling pending transactions, acknowledgements etc. is handled automatically by the Megaco application but the user is free to override the default behaviour by the various configuration possibilities. See megaco:update_user_info/2 and megaco:update_conn_info/2 about the possibilities.

When connections gets broken (that is explicitly by megaco:disconnect/2 or when its controlling process dies) a user callback function is invoked in order to allow the user to re-establish the connection. The internal state of kept messages, re-send timers etc. is not affected by this. A few re-sends will of course fail while the connection is down, but the automatic re-send algorithm does not bother about this and eventually when the connection is up and running the messages will be delivered if the timeouts are set to be long enough. The user has the option of explicitly invoking megaco:cancel/2 to cancel all messages for a connection.

1.3.2 MGC startup call flow

In order to prepare the MGC for the reception of the initial message, hopefully a Service Change Request, the following needs to be done:

- Start the Megaco application.
- Start the MGC user. This may either be done explicitly with megaco:start_user/2 or implicitly by providing the -megaco users configuration parameter.
- Initiate the transport service and provide it with a receive handle obtained from megaco:user_info/2.

When the initial message arrives the transport service forwards it to the protocol engine which automatically sets up the connection and invokes UserMod:handle_connect/2 before it invokes UserMod:handle_trans_request/3 with the Service Change Request like this:

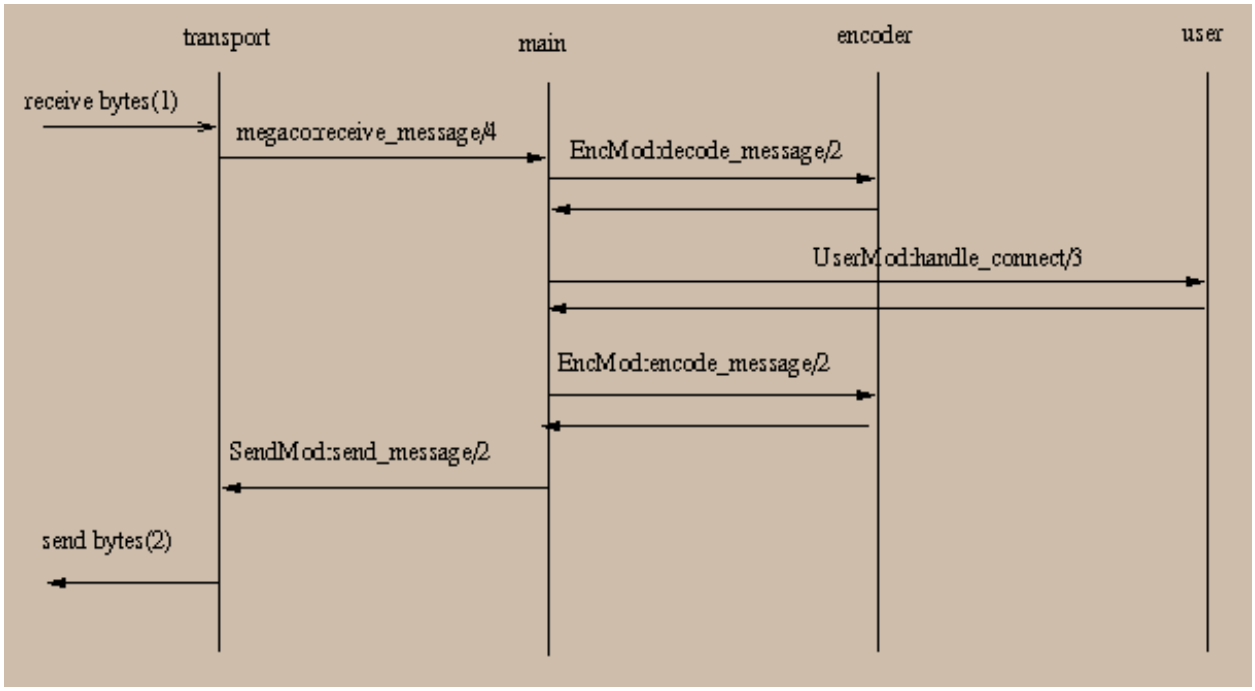


Figure 3.1: MGC Startup Call Flow

1.3.3 MG startup call flow

In order to prepare the MG for the sending of the initial message, hopefully a Service Change Request, the following needs to be done:

- Start the Megaco application.
- Start the MG user. This may either be done explicitly with megaco:start_user/2 or implicitly by providing the -megaco users configuration parameter.
- Initiate the transport service and provide it with a receive handle obtained from megaco:user_info/2.
- Setup a connection to the MGC with megaco:connect/4 and provide it with a receive handle obtained from megaco:user_info/2.

If the MG has been provisioned with the MID of the MGC it can be given as the RemoteMid parameter to megaco:connect/4 and the call flow will look like this:

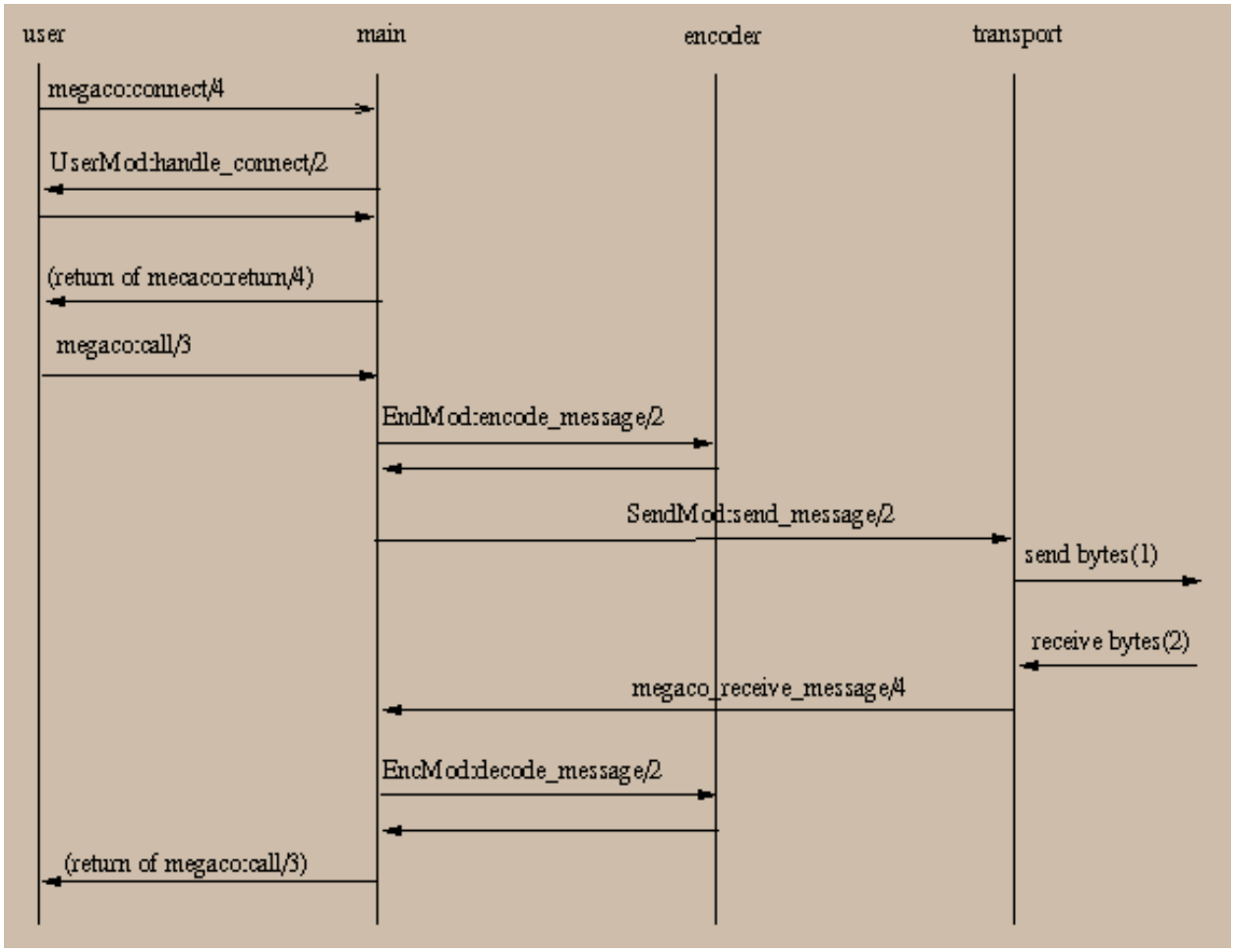


Figure 3.2: MG Startup Call Flow

If the MG cannot be provisioned with the MID of the MGC, the MG can use the atom 'preliminary_mid' as the RemoteMid parameter to megaco:connect/4 and the call flow will look like this:

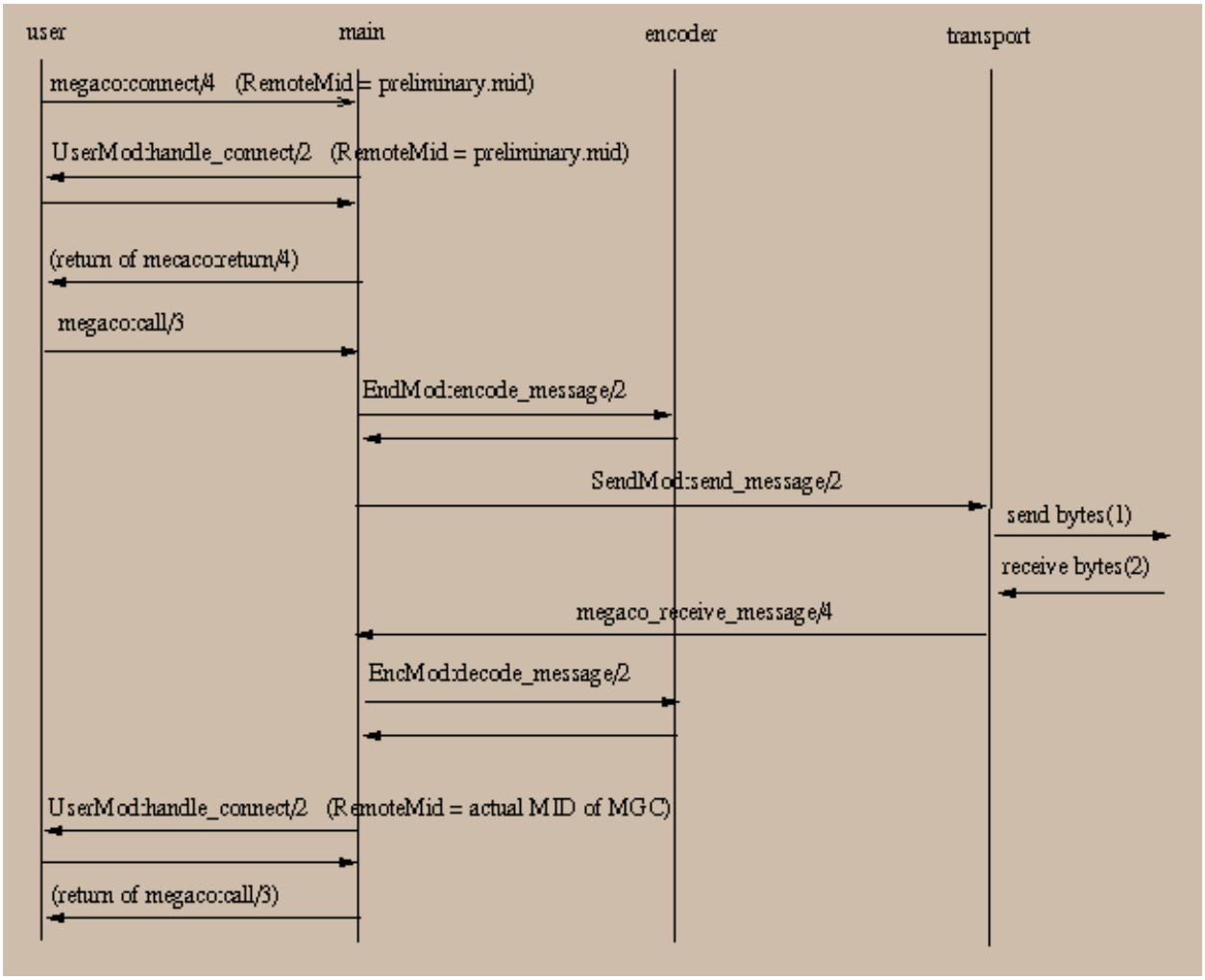


Figure 3.3: MG Startup Call Flow (no MID)

1.3.4 Configuring the Megaco stack

There are three kinds of configuration:

- User info - Information related to megaco users. Read/Write.

  A User is an entity identified by a MID, e.g. a MGC or a MG.

  This information can be retrieved using *megaco:user_info*.
- Connection info - Information regarding connections. Read/Write.

  This information can be retrieved using *megaco:conn_info*.
- System info - System wide information. Read only.

  This information can be retrieved using *megaco:system_info*.

1.3.5 Initial configuration

The initial configuration of the Megaco should be defined in the Erlang system configuration file. The following configured parameters are defined for the Megaco application:

- `users = [{Mid, [user_config()]}].`

  Each user is represented by a tuple with the Mid of the user and a list of config parameters (each parameter is in turn a tuple: `{Item, Value}`).
- `scanner = flex | {Module, Function, Arguments, Modules}`
  - `flex` will result in the start of the flex scanner with default options.
  - The MFA alternative makes it possible for Megaco to start and supervise a scanner written by the user (see `supervisor:start_child` for an explanation of the parameters).

See also *Configuration of text encoding module(s)* for more info.

1.3.6 Changing the configuration

The configuration can be changed during runtime. This is done with the functions *megaco:update_user_info* and *megaco:update_conn_info*

1.3.7 The transaction sender

The transaction sender is a process (one per connection), which handle all transaction sending, if so configured (see *megaco:user_info* and *megaco:conn_info*).

The purpose of the transaction sender is to accumulate transactions for a more efficient message sending. The transactions that are accumulated are transaction request and transaction ack. For transaction ack's the benefit is quite large, since the transactions are small and it is possible to have ranges (which means that transaction acks for transactions 1, 2, 3 and 4 can be sent as a range 1-4 in one transaction ack, instead of four separate transactions).

There are a number of configuration parameter's that control the operation of the transaction sender. In principle, a message with everything stored (ack's and request's) is sent from the process when:

- When `trans_timer` expires.
- When `trans_ack_maxcount` number of ack's has been received.
- When `trans_req_maxcount` number of requests's has been received.
- When the size of all received requests exceeds `trans_req_maxsize`.
- When a reply transaction is sent.
- When a pending transaction is sent.

When something is to be sent, everything is packed into one message, unless the trigger was a reply transaction and the added size of the reply and all the requests is greater then `trans_req_maxsize`, in which case the stored transactions are sent first in a separate message and the reply in another message.

When the transaction sender receives a request which is already "in storage" (indicated by the transaction id) it is assumed to be a resend and everything stored is sent. This could happen if the values of the `trans_timer` and the `request_timer` is not properly chosen.

1.3.8 Segmentation of transaction replies

In version 3 of the megaco standard the Segmentation package was introduced. Simply, this package defines a procedure to segment megaco messages (transaction replies) when using a transport that does not automatically do this (e.g. UDP). See also *version3*.

Although it would be both pointless and counterproductive to use segmentation on a transport that already does this (e.g. TCP), the megaco application does not check this. Instead, it is up to the user to configure this properly.

- Receiving segmented messages:

  This is handled automatically by the megaco application. There is however one thing that need to be configured by the user, the *segment_recv_timer* option.

  Note that the segments are delivered to the user differently depending on which function is used to issue the original request. When issuing the request using the *megaco:cast* function, the segments are delivered to the user via the *handle_trans_reply* callback function one at a time, as they arrive. But this obviously doe not work for the *megaco:call* function. In this case, the segments are accumulated and then delivered all at once as the function returns.
- Sending segmented messages:

  This is also handled automatically by the megaco application. First of all, segmentation is only attempted if so configured, see the *segment_send* option. Secondly, megaco relies on the ability of the used codec to encode action replies, which is the smallest component the megaco application handles when segmenting. Thirdly, the reply will be segmented only if the sum of the size of the action replies (plus an arbitrary message header size) are greater then the specified max message size (see the *max_pdu_size* option). Finally, if segmentation is decided, then each action reply will make up its own (segment) message.

1.4 Internal form and its encodings

This version of the stack is compliant with:

- Megaco/H.248 version 1 (RFC3525) updated according to Implementors Guide version 10-13.
- Megaco/H.248 version 2 as defined by draft-ietf-megaco-h248v2-04 updated according to Implementors Guide version 10-13.
- Megaco/H.248 version 3 as defined by ITU H.248.1 (09/2005).

1.4.1 Internal form of messages

We use the same internal form for both the binary and text encoding. Our internal form of Megaco/H.248 messages is heavily influenced by the internal format used by ASN.1 encoders/decoders:

- "SEQUENCE OF" is represented as a list.
- "CHOICE" is represented as a tagged tuple with size 2.
- "SEQUENCE" is represented as a record, defined in "megaco/include/megaco_message_v1.hrl".
- "OPTIONAL" is represented as an ordinary field in a record which defaults to 'asn1_NOVALUE', meaning that the field has no value.
- "OCTET STRING" is represented as a list of unsigned integers.
- "ENUMERATED" is represented as a single atom.
- "BIT STRING" is represented as a list of atoms.
- "BOOLEAN" is represented as the atom 'true' or 'false'.
- "INTEGER" is represented as an integer.
- "IA5String" is represented as a list of integers, where each integer is the ASCII value of the corresponding character.
- "NULL" is represented as the atom 'NULL'.

In order to fully understand the internal form you must get hold on a ASN.1 specification for the Megaco/H.248 protocol, and apply the rules above. Please, see the documentation of the ASN.1 compiler in Erlang/OTP for more details of the semantics in mapping between ASN.1 and the corresponding internal form.

Observe that the 'TerminationId' record is not used in the internal form. It has been replaced with a megaco_term_id record (defined in "megaco/include/megaco.hrl").

1.4.2 The different encodings

The Megaco/H.248 standard defines both a plain text encoding and a binary encoding (ASN.1 BER) and we have implemented encoders and decoders for both. We do in fact supply five different encoding/decoding modules.

In the text encoding, implementors have the choice of using a mix of short and long keywords. It is also possible to add white spaces to improve readability. We use the term compact for text messages with the shortest possible keywords and no optional white spaces, and the term pretty for a well indented text format using long keywords and an indentation style like the text examples in the Megaco/H.248 specification).

Here follows an example of a text message to give a feeling of the difference between the pretty and compact versions of text messages. First the pretty, well indented version with long keywords:

```
MEGACO/1 [124.124.124.222]
Transaction = 9998 {
        Context = - {
                ServiceChange = ROOT {
                        Services {
                                Method = Restart,
                                ServiceChangeAddress = 55555,
                                Profile = ResGW/1,
                                Reason = "901 Cold Boot"
                        }
                }
        }
}
```

Then the compact version without indentation and with short keywords:

```
!/1 [124.124.124.222]
T=9998{C=-{SC=ROOT{SV{MT=RS,AD=55555,PF=ResGW/1,RE="901 Cold Boot"}}}}
```

And the programmers view of the same message. First a list of ActionRequest records are constructed and then it is sent with one of the send functions in the API:

```
Prof = #'ServiceChangeProfile'{profileName = "resgw", version = 1},
Parm = #'ServiceChangeParm'{serviceChangeMethod  = restart,
                            serviceChangeAddress = {portNumber, 55555},
                            serviceChangeReason  = "901 Cold Boot",
                            serviceChangeProfile = Prof},
Req = #'ServiceChangeRequest'{terminationID = [?megaco_root_termination_id],
                              serviceChangeParms = Parm},
Actions = [#'ActionRequest'{contextId = ?megaco_null_context_id,
                            commandRequests = {serviceChangeReq, Req}}],
megaco:call(ConnHandle, Actions, Config).
```

And finally a print-out of the entire internal form:

```
{'MegacoMessage',
 asn1_NOVALUE,
 {'Message',
  1,
  {ip4Address,{'IP4Address', [124,124,124,222], asn1_NOVALUE}},
  {transactions,
   [
    {transactionRequest,
     {'TransactionRequest',
       9998,
       [{'ActionRequest',
         0,
         asn1_NOVALUE,
         asn1_NOVALUE,
         [
          {'CommandRequest',
           {serviceChangeReq,
            {'ServiceChangeRequest',
             [
              {megaco_term_id, false, ["root"]}],
              {'ServiceChangeParm',
               restart,
               {portNumber, 55555},
               asn1_NOVALUE,
               {'ServiceChangeProfile', "resgw", version = 1},
               "901 MG Cold Boot",
               asn1_NOVALUE,
               asn1_NOVALUE,
               asn1_NOVALUE
              }
             }
           },
           asn1_NOVALUE,
           asn1_NOVALUE
          }
         ]
        }
       ]
     }
    }
   ]
  }
 }
}
```

The following encoding modules are provided:

- megaco_pretty_text_encoder - encodes messages into pretty text format, decodes both pretty as well as compact text.
- megaco_compact_text_encoder - encodes messages into compact text format, decodes both pretty as well as compact text.
- megaco_binary_encoder - encode/decode ASN.1 BER messages. This encoder implements the fastest of the BER encoders/decoders. Recommended binary codec.
- megaco_ber_encoder - encode/decode ASN.1 BER messages.
- megaco_per_encoder - encode/decode ASN.1 PER messages. N.B. that this format is not included in the Megaco standard.
- megaco_erl_dist_encoder - encodes messages into Erlangs distribution format. It is rather verbose but encoding and decoding is blinding fast. N.B. that this format is not included in the Megaco standard.

1.4.3 Configuration of Erlang distribution encoding module

The encoding_config of the megaco_erl_dist_encoder module may be one of these:

- `[]` - Encodes the messages to the standard distribution format. It is rather verbose but encoding and decoding is blinding fast.
- `[megaco_compressed]` - Encodes the messages to the standard distribution format after an internal transformation. It is less verbose, but the total time of the encoding and decoding will on the other hand be somewhat slower (see the *performance* chapter for more info).
- `[{megaco_compressed, Module}]` - Works in the same way as the megaco_compressed config parameter, only here the user provide their own compress module. This module must implement the *megaco_edist_compress* behaviour.
- `[compressed]` - Encodes the messages to a compressed form of the standard distribution format. It is less verbose, but the encoding and decoding will on the other hand be slower.

1.4.4 Configuration of text encoding module(s)

When using text encoding(s), there is actually two different configs controlling what software to use:

- `[]` - An empty list indicates that the erlang scanner should be used.
- `[{flex, port()}]` - Use the flex scanner when decoding (not optimized for SMP). See *initial configuration* for more info.
- `[{flex, ports()}]` - Use the flex scanner when decoding (optimized for SMP). See *initial configuration* for more info.

The Flex scanner is a Megaco scanner written as a linked in driver (in C). There are two ways to get this working:

- Let the Megaco stack start the flex scanner (load the driver).

  To make this happen the megaco stack has to be configured:

  - Add the `{scanner, flex}` (or similar) directive to an Erlang system config file for the megaco app (see *initial configuration* chapter for details).
  - Retrieve the encoding-config using the *system_info* function (with `Item = text_config`).
  - Update the receive handle with the encoding-config (the `encoding_config` field).

  The benefit of this is that Megaco handles the starting, holding and the supervision of the driver and port.
- The Megaco client (user) starts the flex scanner (load the driver).

  When starting the flex scanner a port to the linked in driver is created. This port has to be owned by a process. This process must not die. If it does the port will also terminate. Therefor:

  - Create a permanent process. Make sure this process is supervised (so that if it does die, this will be noticed).
  - Let this process start the flex scanner by calling the `megaco_flex_scanner:start/0,1` function.
  - Retrieve the encoding-config and when initiating the `megaco_receive_handle`, set the field `encoding_config` accordingly.
  - Pass the `megaco_receive_handle` to the transport module.

1.4.5 Configuration of binary encoding module(s)

When using binary encoding, the structure of the termination id's needs to be specified.

- `[native]` - skips the transformation phase, i.e. the decoded message(s) will not be transformed into our internal form.
- `[integer()]` - A list containing the size (the number of bits) of each level. Example: `[3,8,5,8]`.

- `integer()` - Number of one byte (8 bits) levels. N.B. This is currently converted into the previous config. Example: `3` (`[8,8,8]`).

1.4.6 Handling megaco versions

Since the version 3 implemented, in this version of the Megaco application, is preliminary, it is necessary to have a way to handle different version 3 implementations. For this reason the encoding config option `{version3, version3()}` has been introduced. This option, if present, has to be **first** in the encoding config list. Version 1 and 2 codec's ignore this option, if found.

```
version3() -> prev3a | prev3b | prev3c | v3
```

- **prev3a**

  Preliminary version 3, based on TD-33

- **prev3b**

  Preliminary version 3, based on TD-33, but text encoding updated with the final solution for priority in `contextProperty` (which is backward compatible with v2).

- **prev3c**

  Preliminary version 3, based on the final version of the v3-standard, but **excluding** segments!

- **v3**

  Full version 3. Including segmentation. This is the default version 3 variant (i.e. if a version 3 messages is to be encoded/decoded and no version3 encoding config is found, then v3 is assumed).

There are two ways to handle the different megaco encoding versions. Either using **dynamic version detection** (only valid for for incoming messages) or by **explicit version** setting in the connection info.

For incoming messages:

- Dynamic version detection

  Set the protocol version in the megaco_receive_handle to `dynamic` (this is the default).
  This works for those codecs that support partial decode of the version, currently **text**, and ber_bin (`megaco_binary_encoder` and `megaco_ber_bin_encoder`).
  This way the decoder will detect which version is used and then use the proper decoder.

- Explicit version

  Explicitly set the actual protocol version in the megaco_receive_handle.
  Start with version 1. When the initial service change has been performed and version 2 has been negotiated, upgrade the megaco_receive_handle of the transport process (control_pid) to version 2. See *megaco_tcp* and *megaco_udp*.
  Note that if `udp` is used, the same transport process could be used for several connections. This could make upgrading impossible.
  For codecs that does not support partial decode of the version, currently `megaco_ber_encoder`, `megaco_per_encoder` and `megaco_per_bin_encoder`, `dynamic` will revert to version 1.

For outgoing messages:

- Update the connection info protocol_version.
- Override protocol version when sending a message by adding the item `{protocol_version, integer()}` to the Options. See *call* or *cast*.
  Note that this does not effect the messages that are sent autonomously by the stack. They use the protocol_version of the connection info.

1.4.7 Encoder callback functions

The encoder callback interface is defined by the `megaco_encoder` behaviour, see *megaco_encoder*.

1.5 Transport mechanisms

1.5.1 Callback interface

The callback interface of the transport module contains several functions. Some of which are mandatory while others are only optional:

- `send_message` - Send a message. **Mandatory**
- `block` - Block the transport. **Optional**

  This function is usefull for flow control.
- `unblock` - Unblock the transport. **Optional**

For more detail, see the *megaco_transport* behaviour definition.

1.5.2 Examples

The Megaco/H.248 application contains implementations for the two protocols specified by the Megaco/H.248 standard; UDP, see *megaco_udp*, and TCP/TPKT, see *megaco_tcp*.

1.6 Implementation examples

1.6.1 A simple Media Gateway Controller

In megaco/examples/simple/megaco_simple_mgc.erl there is an example of a simple MGC that listens on both text and binary standard ports and is prepared to handle a Service Change Request message to arrive either via TCP/IP or UDP/IP. Messages received on the text port are decoded using a text decoder and messages received on the binary port are decoded using a binary decoder.

The Service Change Reply is encoded in the same way as the request and sent back to the MG with the same transport mechanism UDP/IP or TCP/IP.

After this initial service change message the connection between the MG and MGC is fully established and supervised.

The MGC, with its four listeners, may be started with:

```
cd megaco/examples/simple
erl -pa ../../../megaco/ebin -s megaco_filter -s megaco
megaco_simple_mgc:start().
```

or simply 'gmake mgc'.

The -s megaco_filter option to erl implies, the event tracing mechanism to be enabled and an interactive sequence chart tool to be started. This may be quite useful in order to visualize how your MGC interacts with the Megaco/H.248 protocol stack.

The event traces may alternatively be directed to a file for later analyze. By default the event tracing is disabled, but it may dynamically be enabled without any need for re-compilation of the code.

1.6.2 A simple Media Gateway

In megaco/examples/simple/megaco_simple_mg.erl there is an example of a simple MG that connects to an MGC, sends a Service Change Request and waits synchronously for a reply.

After this initial service change message the connection between the MG and MGC is fully established and supervised.

Assuming that the MGC is started on the local host, four different MG's, using text over TCP/IP, binary over TCP/IP, text over UDP/IP and binary over UDP/IP may be started on the same Erlang node with:

```
cd megaco/examples/simple
erl -pa ../../../megaco/ebin -s megaco_filter -s megaco
megaco_simple_mg:start().
```

or simply 'gmake mg'.

If you "only" want to start a single MG which tries to connect an MG on a host named "baidarka", you may use one of these functions (instead of the megaco_simple_mg:start/0 above):

```
megaco_simple_mg:start_tcp_text("baidarka", []).
megaco_simple_mg:start_tcp_binary("baidarka", []).
megaco_simple_mg:start_udp_text("baidarka", []).
megaco_simple_mg:start_udp_binary("baidarka", []).
```

The -s megaco_filter option to erl implies, the event tracing mechanism to be enabled and an interactive sequence chart tool to be started. This may be quite useful in order to visualize how your MG interacts with the Megaco/H.248 protocol stack.

The event traces may alternatively be directed to a file for later analyze. By default the event tracing is disabled, but it may dynamically be enabled without any need for re-compilation of the code.

1.7 Megaco mib

1.7.1 Intro

The Megaco mib is as of yet not standardized and our implementation is based on **draft-ietf-megaco-mib-04.txt**. Almost all of the mib cannot easily be implemented by the megaco application. Instead these things should be implemented by a user (of the megaco application).

So what part of the mib is implemented? Basically the relevant statistic counters of the **MedGwyGatewayStatsEntry**.

1.7.2 Statistics counters

The implementation of the statistic counters is lightweight. I.e. the statistic counters are handled separately by different entities of the application. For instance our two transport module(s) (see *megaco_tcp* and *megaco_udp*) maintain their own counters and the application engine (see *megaco*) maintain its own counters.

This also means that if a user implement their own transport service then it has to maintain its own statistics.

1.7.3 Distribution

Each megaco application maintains its own set of counters. So in a large (distributed) MG/MGC it could be necessary to collect the statistics from several nodes (each) running the megaco application (only one of them with the transport).

1.8 Performance comparison

1.8.1 Comparison of encoder/decoders

The Megaco/H.248 standard defines both a plain text encoding and a binary encoding (ASN.1 BER) and we have implemented encoders and decoders for both. We do supply a bunch of different encoding/decoding modules and the user may in fact implement their own (like our erl_dist module). Using a non-standard encoding format has its obvious drawbacks, but may be useful in some configurations.

We have made four different measurements of our Erlang/OTP implementation of the Megaco/H.248 protocol stack, in order to compare our different encoders/decoders. The result of each one is summarized in the table below.

The result above are the fastest of these configurations for each codec. The figures presented are the average of all used messages.

For comparison, also included are first, performance figures with megaco (including the measurement software) and asn1 applications hipe-compiled (second figure in the time columns, note that per bin decode had some issues so those figures are not included), and second, performance figures where the flex driver was built as `non-reentrant` flex (third figure in the time columns, only valid for text codecs using the flex-scanner, figures within parenthesis).

Codec and config	Size	Encode	Decode	Total
pretty	336	20 / 13	75 / 40	95 / 53
pretty [flex]	336	20 / 13 / 20	39 / 33 / 38	59 / 46 / 58
compact	181	17 / 10	62 / 35	79 / 45
compact [flex]	181	17 / 10 / 17	37 / 31 / 36	54 / 41 / 53
per bin	91	60 / 29	64 / -	124 / -
per bin [driver]	91	39 / 24	42 / 26	81 / 50
per bin [native]	91	45 / 21	48 / -	93 / -
per bin [driver,native]	91	25 / 15	27 / 18	52 / 33
ber bin	165	32 / 19	38 / 21	70 / 40
ber bin [driver]	165	32 / 19	33 / 20	65 / 39
ber bin [native]	165	17 / 11	25 / 13	42 / 24
ber bin [driver,native]	165	17 / 11	17 / 12	34 / 23
erl_dist	875	5 / 5	10 / 10	15 / 15
erl_dist [megaco_compressed]	405	6 / 4	7 / 4	13 / 8
erl_dist [compressed]	345	47 / 47	20 / 20	67 / 67

erl_dist [megaco_compressed,compressed]	200	34 / 33	11 / 9	45 / 42

Table 8.1: Codec performance

1.8.2 System performance characteristics

This is primarily a way to show the effects of using the reentrant flex scanner instead of the non-reentrant.

As can be seen from the figures above there is no real difference between a non-reentrant and an reentrant flex scanner when it comes to the decode times of an individual message.

The real difference is instead in system characteristics, which is best shown with the mstone1 test.

When running SMP erlang on a multi-core machine the "throughput" is significantly higher. The mstone1 test is an extreme test, but it shows what is gained by using the reentrant flex-scanner.

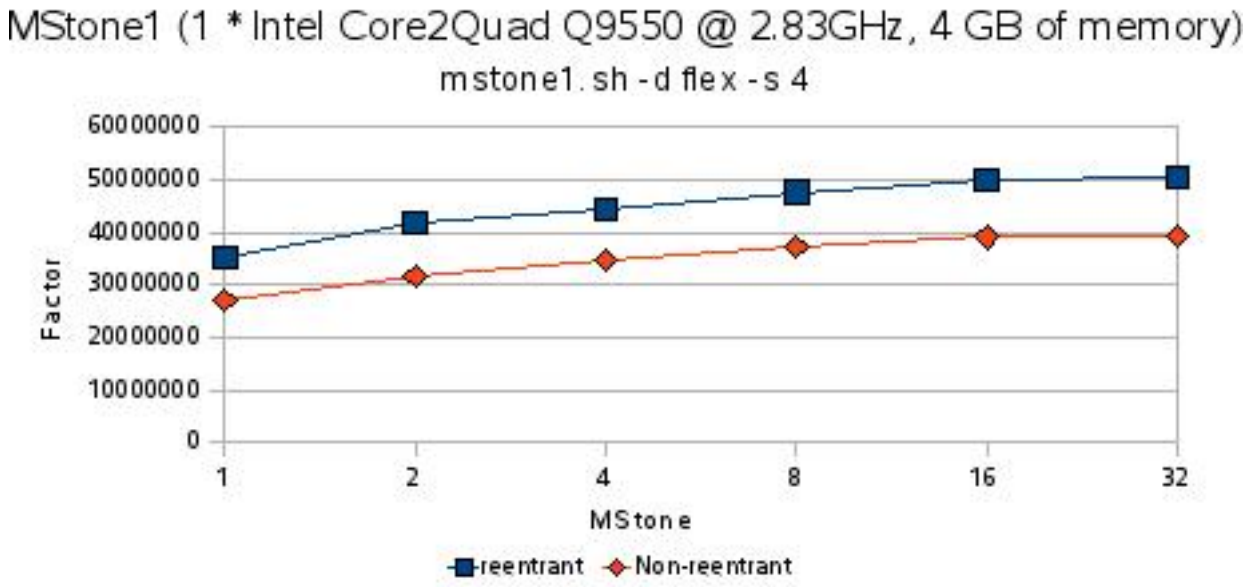


Figure 8.1: MStone1 with mstone1.sh -d flex -s 4

1.8.3 Description of encoders/decoders

In Appendix A of the Megaco/H.248 specification (RFC 3525), there are about 30 messages that shows a representative call flow. We have also added a few extra version 1, version 2 and version 3 messages. We have used these messages as basis for our measurements. Our figures have not been weighted in regard to how frequent the different kinds of messages that are sent between the media gateway and its controller.

The test compares the following encoder/decoders:

- **pretty** - pretty printed text. In the text encoding, the protocol stack implementors have the choice of using a mix of short and long keywords. It is also possible to add white spaces to improve readability. The pretty text encoding utilizes long keywords and an indentation style like the text examples in the Megaco/H.248 specification.
- **compact** - the compact text encoding uses the shortest possible keywords and no optional white spaces.
- **ber** - ASN.1 BER.

- **per** - ASN.1 PER. Not standardized as a valid Megaco/H.248 encoding, but included for the matter of completeness as its encoding is extremely compact.
- **erl_dist** - Erlang's native distribution format. Not standardized as a valid Megaco/H.248 encoding, but included as a reference due to its well known performance characteristics. Erlang is a dynamically typed language and any Erlang data structure may be serialized to the erl_dist format by using built-in functions.

The actual encoded messages have been collected in one directory per encoding type, containing one file per encoded message.

Here follows an example of a text message to give a feeling of the difference between the pretty and compact versions of text messages. First the pretty printed, well indented version with long keywords:

```
MEGACO/1 [124.124.124.222]
  Transaction = 9998 {
    Context = - {
      ServiceChange = ROOT {
        Services {
          Method = Restart,
          ServiceChangeAddress = 55555,
          Profile = ResGW/1,
          Reason = "901 MG Cold Boot"
        }
      }
    }
  }
```

Then the compact text version without indentation and with short keywords:

```
!/1 [124.124.124.222] T=9998{
  C=-{SC=ROOT{SV{MT=RS,AD=55555,PF=ResGW/1,RE="901 MG Cold Boot"}}}}
```

1.8.4 Setup

The measurements has been performed on a HP xw4600 Workstation with a Intel(R) Core(TM)2 Quad CPU Q9550 @ 2.83GHz, with 4 GB memory and running Ubuntu 10.04 x86_64, kernel 2.6.32-22-generic. Software versions was open source OTP R13B04 (megaco-3.14).

1.8.5 Summary

In our measurements we have seen that there are no significant differences in message sizes between ASN.1 BER and the compact text format. Some care should be taken when using the pretty text style (which is used in all the examples included in the protocol specification and preferred during debugging sessions) since the messages can then be quite large. If the message size really is a serious issue, our per encoder should be used, as the ASN.1 PER format is much more compact than all the other alternatives. Its major drawback is that it is has not been approved as a valid Megaco/H.248 message encoding.

When it comes to pure encode/decode performance, it turns out that:

- our fastest binary encoder (ber) is about equal to our fastest text encoder (compact).
- our fastest binary decoder (ber) is about 54% (61%) faster than our fastest text decoder (compact).

If the pure encode/decode performance really is a serious issue, our erl_dist encoder could be used, as the encoding/decoding of the erlang distribution format is much faster than all the other alternatives. Its major drawback is that it is has not been approved as a valid Megaco/H.248 message encoding.

There is no performance advantage of building (and using) a non-reentrant flex scanner over a reentrant flex scanner (if flex supports building such a scanner).

Note:

Please, observe that these performance figures are related to our implementation in Erlang/OTP. Measurements of other implementations using other tools and techniques may of course result in other figures.

1.9 Testing and tools

1.9.1 Tracing

We have instrumented our code in order to enable tracing. Running the application with tracing deactivated, causes a negligible performance overhead (an external call to a function which returns an atom). Activation of tracing does not require any recompilation of the code, since we rely on Erlang/OTP's built in support for dynamic trace activation. In our case tracing of calls to a given external function.

Event traces can be viewed in a generic message sequence chart tool, `et`, or as standard output (events are written to stdio).

See *enable_trace*, *disable_trace* and *set_trace* for more info.

1.9.2 Measurement and transformation

We have included some simple tool(s) for codec measurement (meas), performance tests (mstone1 and mstone2) and message transformation.

The tool(s) are located in the example/meas directory.

Requirement

- Erlang/OTP, version R13B01 or later.
- Version 3.11 or later of **this** application.
- Version 1.6.10 or later of the **asn1** application.
- The flex libraries. Without it, the flex powered codecs cannot be used.

Meas results

The results from the measurement run (meas) is four excel-compatible textfiles:

- decode_time.xls -> Decoding result
- encode_time.xls -> Encoding result
- total_time.xls -> Total (Decoding+encoding) result
- message_size.xls -> Message size

Instruction

The tool contain four things:

- The transformation module
- The measurement (meas) module(s)
- The mstone (mstone1 and mstone2) module(s)
- The basic message file

Message Transformation

The messages used by the different tools are contained in single message package file (see below for more info). The messages in this file is encoded with just one codec. During measurement initiation, the messages are read and then transformed to all codec formats used in the measurement.

The message transformation is done by the transformation module. It is used to transform a set of messages encoded with one codec into the other base codec's.

Measurement(s)

There are two different measurement tools:

- **meas**:

  Used to perform codec measurements. That is, to see what kind of performance can be expected by the different codecs provided by the megaco application.

  The measurement is done by iterating over the decode/encode function for approx 2 seconds per message and counting the number of decodes/encodes.

  Is best run by modifying the meas.sh.skel skeleton script provided by the tool.

  To run it manually do the following:

  ```
  % erl -pa <path-megaco-ebin-dir> -pa <path-to-meas-module-dir>
  Erlang (BEAM) emulator version 5.6 [source]

  Eshell V5.7.1  (abort with ^G)
  1> megaco_codec_meas:start().
  ...
  2> halt().
  ```

  or to make it even easier, assuming a measure shall be done on all the codecs (as above):

  ```
  % erl -noshell -pa <path-megaco-ebin-dir> \\
        -pa <path-to-meas-module-dir> \\
        -s megaco_codec_meas -s init stop
  ```

  When run as above (this will take some time), the measurement process is done as follows:

  ```
  For each codec:
      For each message:
          Read the message from the file
              Detect message version
              Measure decode
                  Measure encode
            Write results, encode, decode and total, to file
  ```

- **mstone1 and mstone2**:

  These are two different SMP performance monitoring tool(s).

  mstone1 creates a process for each codec config supported by the megaco application and let them run for a specific time (all at the same time), encoding and decoding megaco messages. The number of messages processed in total is the mstone1(1) value.

  There are different ways to run the mstone1 tool, e.g. with or without the use of drivers, with **only** flex-empowered configs.

  Is best run by modifying the mstone1.sh.skel skeleton script provided by the tool.

  The **mstone2** is similar to the **mstone1** tool, but in this case, each created process makes only **one** run through the messages and then exits. A soon as a process exits, a new process (with the same config and messages) is created to takes its place. The number of messages processed in total is the mstone2(1) value.

Both these tools use the message package (time_test.msgs) provided with the tool(s), although it can run on any message package as long as it has the same structure.

Message package file

This is simply an erlang compatible text-file with the following structure: `{codec_name(), messages_list()}`.

```
codec_name() = pretty | compact | ber | per | erlang      (how the messages are encoded)
messages_list() = [{message_name(), message()}]
message_name() = atom()
message() = binary()
```

The codec name is the name of the codec with which all messages in the `message_list()` has been encoded.

This file can be `exported` to a file structure by calling the *export_messages* function. This can be usefull if a measurement shall be done with an external tool. Exporting the messages creates a directory tree with the following structure:

```
<message package>/pretty/<message-files>
                  compact/
                  per/
                  ber/<message-files>
                  erlang/
```

The file includes both version 1, 2 and version 3 messages.

Notes

Binary codecs

There are two basic ways to use the binary encodings: With package related name and termination id transformation (the 'native' encoding config) or without. This transformation converts package related names and termination id's to a more convenient internal form (equivalent with the decoded text message).

The transformation is done _after_ the actual decode has been done.

Furthermore, it is possible to make use of a linked in driver that performs some of the decode/encode, decode for ber and encode for per (the 'driver' encoding config).

Therefor in the tests, binary codecs are tested with four different encoding configs to determine exactly how the different options effect the performance: with transformation and without driver ([]), without transformation and without driver ([native]), with transformation and with driver ([driver]) and finally without transformation and with driver ([driver,native]).

Included test messages

Some of these messages are ripped from the call flow examples in an old version of the RFC and others are created to test a specific feature of megaco.

Measurement tool directory name

Be sure **not** no name the directory containing the measurement binaries starting with 'megaco-', e.g. megaco-meas. This will confuse the erlang application loader (erlang applications are named, e.g. megaco-1.0.2).

2 Reference Manual

The Megaco application is a framework for building applications on top of the Megaco/H.248 protocol.

megaco

Erlang module

Interface module for the Megaco application

DATA TYPES

```
megaco_mid() = ip4Address() | ip6Address() |
               domainName() | deviceName() |
               mtpAddress()
ip4Address() = #'IP4Address'{}
ip6Address() = #'IP6Address'{}
domainName() = #'DomainName'{}
deviceName() = pathName()
pathName()   = ia5String(1..64)
mtpAddress() = octetString(2..4)

action_request() = #'ActionRequest'{}
action_reply() = #'ActionReply'{}
error_desc() = #'ErrorDescriptor'{}
transaction_reply() = #'TransactionReply'{}
segment_no() = integer()

resend_indication() = flag | boolean()

property_parm() = #'PropertyParm'{}
property_group() = [property_parm()]
property_groups() = [property_group()]

sdp() = sdp_c() | sdp_o() | sdp_s() | sdp_i() | sdp_u() |
        sdp_e() | sdp_p() | sdp_b() | sdp_z() | sdp_k() |
        sdp_a() | sdp_a_rtpmap() | sdp_a_ptime() |
        sdp_t() | sdp_r() | sdp_m()
sdp_v() = #megaco_sdp_v{} (Protocol version)
sdp_o() = #megaco_sdp_o{} (Owner/creator and session identifier)
sdp_s() = #megaco_sdp_s{} (Session name)
sdp_i() = #megaco_sdp_i{} (Session information)
sdp_u() = #megaco_sdp_u{} (URI of description)
sdp_e() = #megaco_sdp_e{} (Email address)
sdp_p() = #megaco_sdp_p{} (Phone number)
sdp_c() = #megaco_sdp_c{} (Connection information)
sdp_b() = #megaco_sdp_b{} (Bandwidth information)
sdp_k() = #megaco_sdp_k{} (Encryption key)
sdp_a() = #megaco_sdp_a{} (Session attribute)
sdp_a_rtpmap() = #megaco_sdp_a_rtpmap{}
sdp_a_ptime() = #megaco_sdp_a_ptime{}
sdp_a_quality() = #megaco_sdp_a_quality{}
sdp_a_fmtp() = #megaco_sdp_a_fmtp{}
sdp_z() = #megaco_sdp_z{} (Time zone adjustment)
sdp_t() = #megaco_sdp_t{} (Time the session is active)
sdp_r() = #megaco_sdp_r{} (Repeat times)
sdp_m() = #megaco_sdp_m{} (Media name and transport address)
sdp_property_parm() = sdp() | property_parm()
sdp_property_group() = [sdp_property_parm()]
sdp_property_groups() = [sdp_property_group()]

megaco_timer() = infinity | integer() >= 0 | megaco_incr_timer()
megaco_incr_timer() = #megaco_incr_timer{}
```

The record `megaco_incr_timer` contains the following fields:

`wait_for = integer() >= 0`

The actual timer time.

`factor = integer() >= 0`

The factor when calculating the new timer time (`wait_for`).

`incr = integer()`

The increment value when calculating the new timer time (`wait_for`). Note that this value **can** be negative and that a timer restart can therefor lead to a `wait_for` value of zero! It is up to the user to be aware of the consequences of a `wait_for` value of zero.

`max_retries = infinity | infinity_restartable | integer() >= 0`

The maximum number of repetitions of the timer.

There is a special case for this field. When the `max_retries` has the value `infinity_restartable`, it means that the timer is restartable as long as some external event occurs (e.g. receipt of a pending message for instance). But the timer will never be restarted "by itself", i.e. when the timer expires (whatever the timeout time), so does the timer. Whenever the timer is restarted, the timeout time will be calculated in the usual way! Also, as mentioned above, beware the consequences of setting the value to `infinity` if **incr** has been set to an negative value.

Exports

`start() -> ok | {error, Reason}`

Types:

`Reason = term()`

Starts the Megaco application

Users may either explicitly be registered with megaco:start_user/2 and/or be statically configured by setting the application environment variable 'users' to a list of {UserMid, Config} tuples. See the function megaco:start_user/2 for details.

`stop() -> ok | {error, Reason}`

Types:

`Reason = term()`

Stops the Megaco application

`start_user(UserMid, Config) -> ok | {error, Reason}`

Types:

`UserMid = megaco_mid()`
`Config = [{user_info_item(), user_info_value()}]`
`Reason = term()`

Initial configuration of a user

Requires the megaco application to be started. A user is either a Media Gateway (MG) or a Media Gateway Controller (MGC). One Erlang node may host many users.

A user is identified by its UserMid, which must be a legal Megaco MID.

Config is a list of {Item, Value} tuples. See megaco:user_info/2 about which items and values that are valid.

```
stop_user(UserMid) -> ok | {error, Reason}
```

Types:

`UserMid = megaco_mid()`

`Reason = term()`

Delete the configuration of a user

Requires that the user does not have any active connection.

```
user_info(UserMid) -> [{Item, Value}]
user_info(UserMid, Item) -> Value | exit(Reason)
```

Types:

`Handle = user_info_handle()`

`UserMid = megaco_mid()`

`Item = user_info_item()`

`Value = user_info_value()`

`Reason = term()`

Lookup user information

The following Item's are valid:

`connections`

Lists all active connections for this user. Returns a list of megaco_conn_handle records.

`receive_handle`

Construct a megaco_receive_handle record from user config

`trans_id`

Current transaction id.

A positive integer or the atom `undefined_serial` (in case no messages has been sent).

`min_trans_id`

First trans id.

A positive integer, defaults to 1.

`max_trans_id`

Last trans id.

A positive integer or `infinity`, defaults to `infinity`.

`request_timer`

Wait for reply.

The timer is cancelled when a reply is received.

When a pending message is received, the timer is cancelled and the `long_request_timer` is started instead (see below). No resends will be performed from this point (since we now know that the other side has received the request).

When the timer reaches an intermediate expire, the request is resent and the timer is restarted.

When the timer reaches the final expire, either the function `megaco:call` will return with `{error, timeout}` or the callback function `handle_trans_reply` will be called with `UserReply = {error, timeout}` (if `megaco:cast` was used).

A Megaco Timer (see explanation above), defaults to `#megaco_incr_timer{}`.

`long_request_timer`

Wait for reply after having received a pending message.

When the timer reaches an intermediate expire, the timer is restarted.

When a pending message is received, and the `long_request_timer` is **not** "on its final leg", the timer will be restarted, and, if `long_request_resend = true`, the request will be re-sent.

A Megaco Timer (see explanation above), defaults to `60 seconds`.

`long_request_resend`

This option indicates weather the request should be resent until the reply is received, **even** though a pending message has been received.

Normally, after a pending message has been received, the request is not resent (since a pending message is an indication that the request has been received). But since the reply (to the request) can be lost, this behaviour has its values.

It is of course pointless to set this value to **true** unless the `long_request_timer` (see above) is also set to an incremental timer (`#megaco_incr_timer{}`).

A `boolean`, defaults to `false`.

`reply_timer`

Wait for an ack.

When a request is received, some info related to the reply is store internally (e.g. the binary of the reply). This info will live until either an ack is received or this timer expires. For instance, if the same request is received again (e.g. a request with the same transaction id), the (stored) reply will be (re-) sent automatically by megaco.

If the timer is of type `#megaco_incr_timer{}`, then for each intermediate timout, the reply will be resent (this is valid until the ack is received or the timer expires).

A Megaco Timer (see explanation above), defaults to 30000.

`request_keep_alive_timeout`

Specifies the timeout time for the request-keep-alive timer.

This timer is started when the **first** reply to an asynchronous request (issued using the *megaco:cast/3* function) arrives. As long as this timer is running, replies will be delivered via the *handle_trans_reply/4,5* callback function, with their "arrival number" (see `UserReply` of the *handle_trans_reply/4,5* callback function).

Replies arriving after the timer has expired, will be delivered using the *handle_unexpected_trans/3,4* callback function.

The timeout time can have the values: `plain | integer() >= 0`.

Defaults to `plain`.

`call_proxy_gc_timeout`

Timeout time for the call proxy.

When a request is sent using the *call/3* function, a proxy process is started to handle all replies. When the reply has been received and delivered to the user, the proxy process continue to exist for as long as this option specifies. Any received messages, is passed on to the user via the *handle_unexpected_trans* callback function.

The timeout time is in milliseconds. A value of 0 (zero) means that the proxy process will exit directly after the reply has been delivered.

An integer >= 0, defaults to 5000 (= 5 seconds).

`auto_ack`

Automatic send transaction ack when the transaction reply has been received (see `trans_ack` below).

This is used for **three-way-handshake**.

A `boolean`, defaults to `false`.

`trans_ack`

Shall ack's be accumulated or not.

This property is only valid if `auto_ack` is true.

If `auto_ack` is true, then if `trans_ack` is `false`, ack's will be sent immediately. If `trans_ack` is `true`, then ack's will instead be sent to the transaction sender process for accumulation and later sending (see `trans_ack_maxcount`, `trans_req_maxcount`, `trans_req_maxsize`, `trans_ack_maxcount` and `trans_timer`).

See also *transaction sender* for more info.

An `boolean`, defaults to `false`.

`trans_ack_maxcount`

Maximum number of accumulated ack's. At most this many ack's will be accumulated by the transaction sender (if started and configured to accumulate ack's).

See also *transaction sender* for more info.

An `integer`, defaults to 10.

`trans_req`

Shall requests be accumulated or not.

If `trans_req` is `false`, then request(s) will be sent immediately (in its own message).

If `trans_req` is true, then request(s) will instead be sent to the transaction sender process for accumulation and later sending (see `trans_ack_maxcount`, `trans_req_maxcount`, `trans_req_maxsize`, `trans_ack_maxcount` and `trans_timer`).

See also *transaction sender* for more info.

An `boolean`, defaults to `false`.

`trans_req_maxcount`

Maximum number of accumulated requests. At most this many requests will be accumulated by the transaction sender (if started and configured to accumulate requests).

See also *transaction sender* for more info.

An `integer`, defaults to 10.

`trans_req_maxsize`

Maximum size of the accumulated requests. At most this much requests will be accumulated by the transaction sender (if started and configured to accumulate requests).

See also *transaction sender* for more info.

An `integer`, defaults to 2048.

`trans_timer`

Transaction sender timeout time. Has two functions. First, if the value is 0, then transactions will not be accumulated (e.g. the transaction sender process will not be started). Second, if the value is greater then 0 and `auto_ack` and `trans_ack` are both true or if `trans_req` is true, then transaction sender will be started

and transactions (which is depending on the values of `auto_ack`, `trans_ack` and `trans_req`) will be accumulated, for later sending.

See also *transaction sender* for more info.

An `integer`, defaults to 0.

`pending_timer`

Automatically send pending if the timer expires before a transaction reply has been sent. This timer is also called provisional response timer.

A Megaco Timer (see explanation above), defaults to 30000.

`sent_pending_limit`

Sent pending limit (see the MGOriginatedPendingLimit and the MGCOriginatedPendingLimit of the megaco root package). This parameter specifies how many pending messages that can be sent (for a given received transaction request). When the limit is exceeded, the transaction is aborted (see *handle_trans_request_abort*) and an error message is sent to the other side.

Note that this has no effect on the actual sending of pending transactions. This is either implicit (e.g. when receiving a re-sent transaction request for a request which is being processed) or controlled by the pending_timer, see above.

A positive integer or `infinity`, defaults to `infinity`.

`recv_pending_limit`

Receive pending limit (see the MGOriginatedPendingLimit and the MGCOriginatedPendingLimit of the megaco root package). This parameter specifies how many pending messages that can be received (for a sent transaction request). When the limit is exceeded, the transaction is considered lost, and an error returned to the user (through the call-back function **handle_trans_reply**).

A positive integer or `infinity`, defaults to `infinity`.

`send_mod`

Send callback module which exports send_message/2. The function SendMod:send_message(SendHandle, Binary) is invoked when the bytes needs to be transmitted to the remote user.

An `atom`, defaults to `megaco_tcp`.

`encoding_mod`

Encoding callback module which exports encode_message/2 and decode_message/2. The function EncodingMod:encode_message(EncodingConfig, MegacoMessage) is invoked whenever a 'MegacoMessage' record needs to be translated into an Erlang binary. The function EncodingMod:decode_message(EncodingConfig, Binary) is invoked whenever an Erlang binary needs to be translated into a 'MegacoMessage' record.

An `atom`, defaults to `megaco_pretty_text_encoder`.

`encoding_config`

Encoding module config.

A `list`, defaults to `[]`.

`protocol_version`

Actual protocol version.

An `integer`, default is 1.

`strict_version`

Strict version control, i.e. when a message is received, verify that the version is that which was negotiated.

An `boolean`, default is true.

`reply_data`

Default reply data.

Any term, defaults to the atom `undefined`.

`user_mod`

Name of the user callback module. See the the reference manual for megaco_user for more info.

`user_args`

List of extra arguments to the user callback functions. See the the reference manual for megaco_user for more info.

`threaded`

If a received message contains several transaction requests, this option indicates whether the requests should be handled sequentially in the same process (`false`), or if each request should be handled by its own process (`true` i.e. a separate process is spawned for each request).

An `boolean`, defaults to `false`.

`resend_indication`

This option indicates weather the transport module should be told if a message send is a resend or not.

If **false**, megaco messages are sent using the *send_message* function.

If **true**, megaco message **re-sends** are made using the *resend_message* function. The initial message send is still done using the *send_message* function.

The special value **flag** instead indicates that the function *send_message/3* shall be used.

A `resend_indication()`, defaults to `false`.

`segment_reply_ind`

This option specifies if the user shall be notified of received segment replies or not.

See *handle_segment_reply* callback function for more information.

A `boolean`, defaults to `false`.

`segment_recv_timer`

This timer is started when the segment indicated by the `segmentation complete token` is received, but all segments has not yet been received.

When the timer finally expires, a "megaco segments not received" (459) error message is sent to the other side and the user is notified with a `segment timeout UserReply` in either the *handle_trans_reply* callback function or the return value of the *call* function.

A Megaco Timer (see explanation above), defaults to `10000`.

`segment_send`

Shall outgoing messages be segmented or not:

`none`

Do not segment outgoing reply messages. This is useful when either it is known that messages are never to large or that the transport protocol can handle such things on its own (e.g. TCP or SCTP).

`integer() > 0`

Outgoing reply messages will be segmented as needed (see `max_pdu_size` below). This value, K, indicate the outstanding window, i.e. how many segments can be outstanding (not acknowledged) at any given time.

`infinity`

Outgoing reply messages will be segmented as needed (see `max_pdu_size` below). Segment messages are sent all at once (i.e. no acknowledgement awaited before sending the next segment).

Defaults to `none`.

`max_pdu_size`

Max message size. If the encoded message (PDU) exceeds this size, the message should be segmented, and then encoded.

A positive integer or `infinity`, defaults to `infinity`.

```
update_user_info(UserMid, Item, Value) -> ok | {error, Reason}
```

Types:

```
UserMid = megaco_mid()
Item = user_info_item()
Value = user_info_value()
Reason = term()
```

Update information about a user

Requires that the user is started. See megaco:user_info/2 about which items and values that are valid.

```
conn_info(ConnHandle) -> [{Item, Value}]
conn_info(ConnHandle, Item) -> Value | exit(Reason)
```

Types:

```
ConnHandle = #megaco_conn_handle{}
Item = conn_info_item()
Value = conn_info_value()
Reason = {no_such_connection, ConnHandle} | term()
```

Lookup information about an active connection

Requires that the connection is active.

`control_pid`

The process identifier of the controlling process for a connection.

`send_handle`

Opaque send handle whose contents is internal for the send module. May be any term.

`local_mid`

The local mid (of the connection, i.e. the own mid). `megaco_mid()`.

`remote_mid`

The remote mid (of the connection). `megaco_mid()`.

`receive_handle`

Construct a megaco_receive_handle record.

`trans_id`

Next transaction id. A positive integer or the atom `undefined_serial` (only in case of error).

Note that transaction id's are (currently) maintained on a per user basis so there is no way to be sure that the value returned will actually be used for a transaction sent on this connection (in case a user has several connections, which is not at all unlikely).

`max_trans_id`

Last trans id.

A positive integer or `infinity`, defaults to `infinity`.

`request_timer`

Wait for reply.

The timer is cancelled when a reply is received.

When a pending message is received, the timer is cancelled and the `long_request_timer` is started instead (see below). No resends will be performed from this point (since we now know that the other side has received the request).

When the timer reaches an intermediate expire, the request is resent and the timer is restarted.

When the timer reaches the final expire, either the function `megaco:call` will return with `{error, timeout}` or the callback function `handle_trans_reply` will be called with `UserReply = {error, timeout}` (if `megaco:cast` was used).

A Megaco Timer (see explanation above), defaults to #megaco_incr_timer{}.

`long_request_timer`

Wait for reply after having received a pending message.

When the timer reaches an intermediate expire, the timer restarted.

When a pending message is received, and the `long_request_timer` is **not** "on its final leg", the timer will be restarted, and, if `long_request_resend = true`, the request will be re-sent.

A Megaco Timer (see explanation above), defaults to `60 seconds`.

`request_keep_alive_timeout`

Specifies the timeout time for the request-keep-alive timer.

This timer is started when the **first** reply to an asynchronous request (issued using the *megaco:cast/3* function) arrives. As long as this timer is running, replies will be delivered via the *handle_trans_reply/4,5* callback function, with their "arrival number" (see `UserReply` of the *handle_trans_reply/4,5* callback function).

Replies arriving after the timer has expired, will be delivered using the *handle_unexpected_trans/3,4* callback function.

The timeout time can have the values: `plain | integer() >= 0`.

Defaults to `plain`.

`long_request_resend`

This option indicates weather the request should be resent until the reply is received, **even** though a pending message has been received.

Normally, after a pending message has been received, the request is not resent (since a pending message is an indication that the request has been received). But since the reply (to the request) can be lost, this behaviour has its values.

It is of course pointless to set this value to **true** unless the `long_request_timer` (see above) is also set to an incremental timer (`#megaco_incr_timer{}`).

A `boolean`, defaults to `false`.

`reply_timer`

Wait for an ack.

When a request is received, some info related to the reply is store internally (e.g. the binary of the reply). This info will live until either an ack is received or this timer expires. For instance, if the same request is received again (e.g. a request with the same transaction id), the (stored) reply will be (re-) sent automatically by megaco.

If the timer is of type `#megaco_incr_timer{}`, then for each intermediate timout, the reply will be resent (this is valid until the ack is received or the timer expires).

A Megaco Timer (see explanation above), defaults to 30000.

`call_proxy_gc_timeout`

Timeout time for the call proxy.

When a request is sent using the *call/3* function, a proxy process is started to handle all replies. When the reply has been received and delivered to the user, the proxy process continue to exist for as long as this option specifies. Any received messages, is passed on to the user via the *handle_unexpected_trans* callback function.

The timeout time is in milliseconds. A value of 0 (zero) means that the proxy process will exit directly after the reply has been delivered.

An integer >= 0, defaults to 5000 (= 5 seconds).

`auto_ack`

Automatic send transaction ack when the transaction reply has been received (see `trans_ack` below).

This is used for **three-way-handshake**.

A `boolean`, defaults to `false`.

`trans_ack`

Shall ack's be accumulated or not.

This property is only valid if `auto_ack` is true.

If `auto_ack` is true, then if `trans_ack` is `false`, ack's will be sent immediately. If `trans_ack` is `true`, then ack's will instead be sent to the transaction sender process for accumulation and later sending (see `trans_ack_maxcount`, `trans_req_maxcount`, `trans_req_maxsize`, `trans_ack_maxcount` and `trans_timer`).

See also *transaction sender* for more info.

An `boolean`, defaults to `false`.

`trans_ack_maxcount`

Maximum number of accumulated ack's. At most this many ack's will be accumulated by the transaction sender (if started and configured to accumulate ack's).

See also *transaction sender* for more info.

An integer, defaults to 10.

`trans_req`

Shall requests be accumulated or not.

If `trans_req` is `false`, then request(s) will be sent immediately (in its own message).

If `trans_req` is true, then request(s) will instead be sent to the transaction sender process for accumulation and later sending (see `trans_ack_maxcount`, `trans_req_maxcount`, `trans_req_maxsize`, `trans_ack_maxcount` and `trans_timer`).

See also *transaction sender* for more info.

An `boolean`, defaults to `false`.

`trans_req_maxcount`

Maximum number of accumulated requests. At most this many requests will be accumulated by the transaction sender (if started and configured to accumulate requests).

See also *transaction sender* for more info.

An `integer`, defaults to 10.

`trans_req_maxsize`

Maximum size of the accumulated requests. At most this much requests will be accumulated by the transaction sender (if started and configured to accumulate requests).

See also *transaction sender* for more info.

An `integer`, defaults to 2048.

`trans_timer`

Transaction sender timeout time. Has two functions. First, if the value is 0, then transactions will not be accumulated (e.g. the transaction sender process will not be started). Second, if the value is greater then 0 and `auto_ack` and `trans_ack` is true or if `trans_req` is true, then transaction sender will be started and transactions (which is depending on the values of `auto_ack`, `trans_ack` and `trans_req`) will be accumulated, for later sending.

See also *transaction sender* for more info.

An `integer`, defaults to 0.

`pending_timer`

Automatic send transaction pending if the timer expires before a transaction reply has been sent. This timer is also called provisional response timer.

A Megaco Timer (see explanation above), defaults to 30000.

`sent_pending_limit`

Sent pending limit (see the MGOriginatedPendingLimit and the MGCOriginatedPendingLimit of the megaco root package). This parameter specifies how many pending messages that can be sent (for a given received transaction request). When the limit is exceeded, the transaction is aborted (see *handle_trans_request_abort*) and an error message is sent to the other side.

Note that this has no effect on the actual sending of pending transactions. This is either implicit (e.g. when receiving a re-sent transaction request for a request which is being processed) or controlled by the pending_timer, see above.

A positive integer or `infinity`, defaults to `infinity`.

`recv_pending_limit`

Receive pending limit (see the MGOriginatedPendingLimit and the MGCOriginatedPendingLimit of the megaco root package). This parameter specifies how many pending messages that can be received (for a sent transaction request). When the limit is exceeded, the transaction is considered lost, and an error returned to the user (through the call-back function **handle_trans_reply**).

A positive integer or `infinity`, defaults to `infinity`.

`send_mod`

Send callback module which exports send_message/2. The function SendMod:send_message(SendHandle, Binary) is invoked when the bytes needs to be transmitted to the remote user.

An `atom`, defaults to `megaco_tcp`.

`encoding_mod`

Encoding callback module which exports encode_message/2 and decode_message/2. The function EncodingMod:encode_message(EncodingConfig, MegacoMessage) is invoked whenever a 'MegacoMessage' record needs to be translated into an Erlang binary. The function EncodingMod:decode_message(EncodingConfig, Binary) is invoked whenever an Erlang binary needs to be translated into a 'MegacoMessage' record.

An `atom`, defaults to `megaco_pretty_text_encoder`.

`encoding_config`

Encoding module config.

A `list`, defaults to [].

`protocol_version`

Actual protocol version.

An positive integer, Current default is 1.

`strict_version`

Strict version control, i.e. when a message is received, verify that the version is that which was negotiated.

An `boolean`, default is true.

`reply_data`

Default reply data.

Any term, defaults to the atom `undefined`.

`threaded`

If a received message contains several transaction requests, this option indicates whether the requests should be handled sequentially in the same process (`false`), or if each request should be handled by its own process (`true` i.e. a separate process is spawned for each request).

An `boolean`, defaults to `false`.

`resend_indication`

This option indicates weather the transport module should be told if a message send is a resend or not.

If **false**, megaco messages are sent using the *send_message/2* function.

If **true**, megaco message **re-sends** are made using the *resend_message* function. The initial message send is still done using the *send_message* function.

The special value **flag** instead indicates that the function *send_message/3* shall be used.

A `resend_indication()`, defaults to `false`.

`segment_reply_ind`

This option specifies if the user shall be notified of received segment replies or not.

See *handle_segment_reply* callback function for more information.

A `boolean`, defaults to `false`.

`segment_recv_timer`

This timer is started when the segment indicated by the `segmentation complete token` (e.g. the last of the segment which makes up the reply) is received, but all segments has not yet been received.

When the timer finally expires, a "megaco segments not received" (459) error message is sent to the other side and the user is notified with a `segment timeout UserReply` in either the *handle_trans_reply* callback function or the return value of the *call* function.

A Megaco Timer (see explanation above), defaults to `10000`.

`segment_send`

Shall outgoing messages be segmented or not:

`none`

Do not segment outgoing reply messages. This is useful when either it is known that messages are never to large or that the transport protocol can handle such things on its own (e.g. TCP or SCTP).

`integer() > 0`

Outgoing reply messages will be segmented as needed (see `max_pdu_size` below). This value, K, indicate the outstanding window, i.e. how many segments can be outstanding (not acknowledged) at any given time.

`infinity`

Outgoing reply messages will be segmented as needed (see `max_pdu_size` below). Segment messages are sent all at once (i.e. no acknowledgement awaited before sending the next segment).

Defaults to `none`.

`max_pdu_size`

Max message size. If the encoded message (PDU) exceeds this size, the message should be segmented, and then encoded.

A positive integer or `infinity`, defaults to `infinity`.

```
update_conn_info(ConnHandle, Item, Value) -> ok | {error, Reason}
```

Types:

```
ConnHandle = #megaco_conn_handle{}
Item = conn_info_item()
Value = conn_info_value()
Reason = term()
```

Update information about an active connection

Requires that the connection is activated. See megaco:conn_info/2 about which items and values that are valid.

```
system_info() -> [{Item, Value}] | exit(Reason)
system_info(Item) -> Value | exit(Reason)
```

Types:

```
Item = system_info_item()
```

Lookup system information

The following items are valid:

`text_config`

The text encoding config.

`connections`

Lists all active connections. Returns a list of megaco_conn_handle records.

`users`

Lists all active users. Returns a list of megaco_mid()'s.

`n_active_requests`

Returns an integer representing the number of requests that has originated from this Erlang node and still are active (and therefore consumes system resources).

`n_active_replies`

Returns an integer representing the number of replies that has originated from this Erlang node and still are active (and therefore consumes system resources).

`n_active_connections`

Returns an integer representing the number of active connections.

`info() -> Info`

Types:

```
Info = [{Key, Value}]
```

This function produces a list of information about the megaco application. Such as users and their config, connections and their config, statistics and so on.

This information can be produced by the functions *user_info*, *conn_info*, *system_info* and *get_stats* but this is a simple way to get it all at once.

`connect(ReceiveHandle, RemoteMid, SendHandle, ControlPid) -> {ok, ConnHandle} | {error, Reason}`

`connect(ReceiveHandle, RemoteMid, SendHandle, ControlPid, Extra) -> {ok, ConnHandle} | {error, Reason}`

Types:

```
ReceiveHandle = #megaco_receive_handle{}
RemoteMid = preliminary_mid | megaco_mid()
SendHandle = term()
ControlPid = pid()
ConnHandle = #megaco_conn_handle{}
Reason = connect_reason() | handle_connect_reason() | term()
connect_reason() = {no_such_user, LocalMid} | {already_connected,
ConnHandle} | term()
handle_connect_error() = {connection_refused, ConnData, ErrorInfo} |
term()
LocalMid = megaco_mid()
ConnData = term()
ErrorInfo = term()
Extra = term()
```

Establish a "virtual" connection

Activates a connection to a remote user. When this is done the connection can be used to send messages (with SendMod:send_message/2). The ControlPid is the identifier of a process that controls the connection. That process

will be supervised and if it dies, this will be detected and the UserMod:handle_disconnect/2 callback function will be invoked. See the megaco_user module for more info about the callback arguments. The connection may also explicitly be deactivated by invoking megaco:disconnect/2.

The ControlPid may be the identity of a process residing on another Erlang node. This is useful when you want to distribute a user over several Erlang nodes. In such a case one of the nodes has the physical connection. When a user residing on one of the other nodes needs to send a request (with megaco:call/3 or megaco:cast/3), the message will encoded on the originating Erlang node, and then be forwarded to the node with the physical connection. When the reply arrives, it will be forwarded back to the originator. The distributed connection may explicitly be deactivated by a local call to megaco:disconnect/2 or implicitly when the physical connection is deactivated (with megaco:disconnect/2, killing the controlling process, halting the other node, ...).

The call of this function will trigger the callback function UserMod:handle_connect/2 to be invoked. See the megaco_user module for more info about the callback arguments.

A connection may be established in several ways:

`provisioned MID`

The MG may explicitly invoke megaco:connect/4 and use a provisioned MID of the MGC as the RemoteMid.

`upgrade preliminary MID`

The MG may explicitly invoke megaco:connect/4 with the atom 'preliminary_mid' as a temporary MID of the MGC, send an intial message, the Service Change Request, to the MGC and then wait for an initial message, the Service Change Reply. When the reply arrives, the Megaco application will pick the MID of the MGC from the message header and automatically upgrade the connection to be a "normal" connection. By using this method of establishing the connection, the callback function UserMod:handle_connect/2 to be invoked twice. First with a ConnHandle with the remote_mid-field set to preliminary_mid, and then when the connection upgrade is done with the remote_mid-field set to the actual MID of the MGC.

`automatic`

When the MGC receives its first message, the Service Change Request, the Megaco application will automatically establish the connection by using the MG MID found in the message header as remote mid.

`distributed`

When a user (MG/MGC) is distributed over several nodes, it is required that the node hosting the connection already has activated the connection and that it is in the "normal" state. The RemoteMid must be a real Megaco MID and not a preliminary_mid.

An initial megaco_receive_handle record may be obtained with megaco:user_info(UserMid, receive_handle)

The send handle is provided by the preferred transport module, e.g. megaco_tcp, megaco_udp. Read the documentation about each transport module about the details.

The connect is done in two steps: first an internal `connection setup` and then by calling the user *handle_connect* callback function. The first step could result in an error with `Reason = connect_reason()` and the second an error with `Reason = handle_connect_reason()`:

`connect_reason()`

An error with this reason is generated by the megaco application itself.

`handle_connect_reason()`

An error with this reason is caused by the user *handle_connect* callback function either returning an error or an invalid value.

`Extra` can be any `term()` except the atom `ignore_extra`. It is passed (back) to the user via the callback function *handle_connect/3*.

```
disconnect(ConnHandle, DiscoReason) -> ok | {error, ErrReason}
```

Types:

ConnHandle = conn_handle()

DiscoReason = term()

ErrReason = term()

Tear down a "virtual" connection

Causes the UserMod:handle_disconnect/2 callback function to be invoked. See the megaco_user module for more info about the callback arguments.

```
call(ConnHandle, Actions, Options) -> {ProtocolVersion, UserReply}
```

Types:

ConnHandle = conn_handle()

Actions = action_reqs() | [action_reqs()]

action_reqs() = binary() | [action_request()]

Options = [send_option()]

send_option() = {request_timer, megaco_timer()} | {long_request_timer, megaco_timer()} | {send_handle, term()} | {protocol_version, integer()} | {call_proxy_gc_timeout, call_proxy_gc_timeout()}

ProtocolVersion = integer()

UserReply = user_reply() | [user_reply()]

user_reply() = success() | failure()

success() = {ok, result()} | {ok, result(), extra()}

result() = message_result() | segment_result()

message_result() = action_reps()

segment_result() = segments_ok()

failure() = {error, reason()} | {error, reason(), extra()}

reason() = message_reason() | segment_reason() | user_cancel_reason() | send_reason() | other_reason()

message_reason() = error_desc()

segment_reason() = {segment, segments_ok(), segments_err()} | {segment_timeout, missing_segments(), segments_ok(), segments_err()}

segments_ok() = [segment_ok()]

segment_ok() = {segment_no(), action_reps()}

segments_err() = [segment_err()]

segment_err() = {segment_no(), error_desc()}

missing_segments() = [segment_no()]

user_cancel_reason() = {user_cancel, reason_for_user_cancel()}

reason_for_user_cancel() = term()

send_reason() = send_cancelled_reason() | send_failed_reason()

send_cancelled_reason() = {send_message_cancelled, reason_for_send_cancel()}

reason_for_send_cancel() = term()

send_failed_reason() = {send_message_failed, reason_for_send_failure()}

reason_for_send_failure() = term()

```
other_reason() = {wrong_mid, WrongMid, RightMid, TR} | term()
WrongMid = mid()
RightMid = mid()
TR = transaction_reply()
action_reps() = [action_reply()]
call_proxy_gc_timeout() = integer() >= 0
extra() = term()
```

Sends one or more transaction request(s) and waits for the reply.

When sending one transaction in a message, `Actions` should be `action_reqs()` (`UserReply` will then be `user_reply()`). When sending several transactions in a message, `Actions` should be `[action_reqs()]` (`UserReply` will then be `[user_reply()]`). Each element of the list is part of one transaction.

For some of **our** codecs (not binary), it is also possible to pre-encode the actions, in which case `Actions` will be either a `binary()` or `[binary()]`.

The function returns when the reply arrives, when the request timer eventually times out or when the outstanding requests are explicitly cancelled.

The default values of the send options are obtained by `megaco:conn_info(ConnHandle, Item)`. But the send options above, may explicitly be overridden.

The `ProtocolVersion` version is the version actually encoded in the reply message.

At `success()`, the `UserReply` contains a list of 'ActionReply' records possibly containing error indications.

A `message_error()`, indicates that the remote user has replied with an explicit transactionError.

A `user_cancel_error()`, indicates that the request has been canceled by the user. `reason_for_user_cancel()` is the reason given in the call to the *cancel* function.

A `send_error()`, indicates that the send function of the megaco transport callback module failed to send the request. There are two separate cases: `send_cancelled_reason()` and `send_failed_reason()`. The first is the result of the send function returning `{cancel, Reason}` and the second is some other kind of erroneous return value. See the *send_message* function for more info.

An `other_error()`, indicates some other error such as timeout.

For more info about the `extra()` part of the result, see the *note* in the user callback module documentation.

```
cast(ConnHandle, Actions, Options) -> ok | {error, Reason}
```

Types:

```
ConnHandle = conn_handle()
Actions = action_reqs() | [action_reqs()]
action_reqs() = binary() | [action_request()]
Options = [send_option()]
send_option() = {request_keep_alive_timeout, request_keep_alive_timeout()}
| {request_timer, megaco_timer()} | {long_request_timer, megaco_timer()}
| {send_handle, term()} | {reply_data, reply_data()} | {protocol_version,
integer()}
request_keep_alive_timeout() = plain | integer() >= 0
Reason = term()
```

Sends one or more transaction request(s) but does NOT wait for a reply

When sending one transaction in a message, `Actions` should be `action_reqs()`. When sending several transactions in a message, `Actions` should be `[action_reqs()]`. Each element of the list is part of one transaction.

For some of **our** codecs (not binary), it is also possible to pre-encode the actions, in which case `Actions` will be either a `binary()` or `[binary()]`.

The default values of the send options are obtained by megaco:conn_info(ConnHandle, Item). But the send options above, may explicitly be overridden.

The ProtocolVersion version is the version actually encoded in the reply message.

The callback function UserMod:handle_trans_reply/4 is invoked when the reply arrives, when the request timer eventually times out or when the outstanding requests are explicitly cancelled. See the megaco_user module for more info about the callback arguments.

Given as UserData argument to UserMod:handle_trans_reply/4.

```
encode_actions(ConnHandle, Actions, Options) -> {ok, BinOrBins} | {error,
Reason}
```

Types:

```
ConnHandle = conn_handle()
Actions = action_reqs() | [action_reqs()]
action_reqs() = [#'ActionRequest'{}]
Options = [send_option()]
send_option() = {request_timer, megaco_timer()} | {long_request_timer,
megaco_timer()} | {send_handle, term()} | {protocol_version, integer()}
BinOrBins = binary() | [binary()]
Reason = term()
```

Encodes lists of action requests for one or more transaction request(s).

When encoding action requests for one transaction, `Actions` should be `action_reqs()`. When encoding action requests for several transactions, `Actions` should be `[action_reqs()]`. Each element of the list is part of one transaction.

```
token_tag2string(Tag) -> Result
token_tag2string(Tag, EncoderMod) -> Result
token_tag2string(Tag, EncoderMod, Version) -> Result
```

Types:

```
Tag = atom()
EncoderMod = pretty | compact | encoder_module()
encoder_module() = megaco_pretty_text_encoder |
megaco_compact_text_encoder | atom()
Version = int_version() | atom_version()
int_version() = 1 | 2 | 3
atom_version() = v1 | v2 | v3 | prev3c | prev3b
Result = string() | {error, Reason}
Reason = term()
```

Convert a token tag to a string

If no encoder module is given, the default is used (which is pretty).

If no or an unknown version is given, the **best** version is used (which is v3).

If no match is found for `Tag`, `Result` will be the empty string (`[]`).

```
cancel(ConnHandle, CancelReason) -> ok | {error, ErrReason}
```

Types:

`ConnHandle = conn_handle()`
`CancelReason = term()`
`ErrReason = term()`

Cancel all outstanding messages for this connection

This causes outstanding megaco:call/3 requests to return. The callback functions UserMod:handle_reply/4 and UserMod:handle_trans_ack/4 are also invoked where it applies. See the megaco_user module for more info about the callback arguments.

```
process_received_message(ReceiveHandle, ControlPid, SendHandle, BinMsg) -> ok
process_received_message(ReceiveHandle, ControlPid, SendHandle, BinMsg,
Extra) -> ok
```

Types:

`ReceiveHandle = #megaco_receive_handle{}`
`ControlPid = pid()`
`SendHandle = term()`
`BinMsg = binary()`
`Extra = term()`

Process a received message

This function is intended to be invoked by some transport modules when get an incoming message. Which transport that actually is used is up to the user to choose.

The message is delivered as an Erlang binary and is decoded by the encoding module stated in the receive handle together with its encoding config (also in the receive handle). Depending of the outcome of the decoding various callback functions will be invoked. See megaco_user for more info about the callback arguments.

The argument `Extra` is just an opaque data structure passed to the user via the callback functions in the *user callback module*. Note however that if `Extra` has the value `extra_undefined` the argument will be ignored (same as if `process_received_message/4` had been called). See the documentation for the behaviour of the callback module, *megaco_user*, for more info.

Note that all processing is done in the context of the calling process. A transport module could call this function via one of the `spawn` functions (e.g. `spawn_opt`). See also `receive_message/4,5`.

If the message cannot be decoded the following callback function will be invoked:

- UserMod:handle_syntax_error/3

If the decoded message instead of transactions contains a message error, the following callback function will be invoked:

- UserMod:handle_message_error/3

If the decoded message happens to be received before the connection is established, a new "virtual" connection is established. This is typically the case for the Media Gateway Controller (MGC) upon the first Service Change. When this occurs the following callback function will be invoked:

- UserMod:handle_connect/2

For each transaction request in the decoded message the following callback function will be invoked:

- UserMod:handle_trans_request/3

For each transaction reply in the decoded message the reply is returned to the user. Either the originating function megaco:call/3 will return. Or in case the originating function was megaco:case/3 the following callback function will be invoked:

- UserMod:handle_trans_reply/4

When a transaction acknowledgement is received it is possible that user has decided not to bother about the acknowledgement. But in case the return value from UserMod:handle_trans_request/3 indicates that the acknowledgement is important the following callback function will be invoked:

- UserMod:handle_trans_ack/4

See the megaco_user module for more info about the callback arguments.

```
receive_message(ReceiveHandle, ControlPid, SendHandle, BinMsg) -> ok
receive_message(ReceiveHandle, ControlPid, SendHandle, BinMsg, Extra) -> ok
```

Types:

```
ReceiveHandle = #megaco_receive_handle{}
ControlPid = pid()
SendHandle = term()
BinMsg = binary()
Extra = term()
```

Process a received message

This is a callback function intended to be invoked by some transport modules when get an incoming message. Which transport that actually is used is up to the user to choose.

In principle, this function calls the `process_received_message/4` function via a `spawn` to perform the actual processing.

For further information see the *process_received_message/4* function.

```
parse_digit_map(DigitMapBody) -> {ok, ParsedDigitMap} | {error, Reason}
```

Types:

```
DigitMapBody = string()
ParsedDigitMap = parsed_digit_map()
parsed_digit_map() = term()
Reason = term()
```

Parses a digit map body

Parses a digit map body, represented as a list of characters, into a list of state transitions suited to be evaluated by megaco:eval_digit_map/1,2.

```
eval_digit_map(DigitMap) -> {ok, MatchResult} | {error, Reason}
eval_digit_map(DigitMap, Timers) -> {ok, MatchResult} | {error, Reason}
```

Types:

```
DigitMap = #'DigitMapValue'{} | parsed_digit_map()
parsed_digit_map() = term()
ParsedDigitMap = term()
```

```
Timers = ignore() | reject()
ignore() = ignore | {ignore, digit_map_value()}
reject() = reject | {reject, digit_map_value()} | digit_map_value()
MatchResult = {Kind, Letters} | {Kind, Letters, Extra}
Kind = kind()
kind() = full | unambiguous
Letters = [letter()]
letter() = $0..$9 | $a .. $k
Extra = letter()
Reason = term()
```

Collect digit map letters according to the digit map.

When evaluating a digit map, a state machine waits for timeouts and letters reported by megaco:report_digit_event/2. The length of the various timeouts are defined in the digit_map_value() record.

When a complete sequence of valid events has been received, the result is returned as a list of letters.

There are two options for handling syntax errors (that is when an unexpected event is received when the digit map evaluator is expecting some other event). The unexpected events may either be ignored or rejected. The latter means that the evaluation is aborted and an error is returned.

```
report_digit_event(DigitMapEvalPid, Events) -> ok | {error, Reason}
```

Types:

```
DigitMapEvalPid = pid()
Events = Event | [Event]
Event = letter() | pause() | cancel()
letter() = $0..$9 | $a .. $k | $A .. $K
pause() = one_second() | ten_seconds()
one_second() = $s | $S
ten_seconds() = $l | $L
cancel() = $z | $Z | cancel
Reason = term()
```

Send one or more events to the event collector process.

Send one or more events to a process that is evaluating a digit map, that is a process that is executing megaco:eval_digit_map/1,2.

Note that the events `$s | $S`, `l | $L` and `$z | $Z` has nothing to do with the timers using the same characters.

```
test_digit_event(DigitMap, Events) -> {ok, Kind, Letters} | {error, Reason}
```

Types:

```
DigitMap = #'DigitMapValue'{} | parsed_digit_map()
parsed_digit_map() = term()
ParsedDigitMap = term()
Timers = ignore() | reject()
ignore() = ignore | {ignore, digit_map_value()}
reject() = reject | {reject, digit_map_value()} | digit_map_value()
DigitMapEvalPid = pid()
```

```
Events = Event | [Event]
Event = letter() | pause() | cancel()
Kind = kind()
kind() = full | unambiguous
Letters = [letter()]
letter() = $0..$9 | $a .. $k | $A .. $K
pause() = one_second() | ten_seconds()
one_second() = $s | $S
ten_seconds() = $l | $L
cancel () = $z | $Z | cancel
Reason = term()
```

Feed digit map collector with events and return the result

This function starts the evaluation of a digit map with megaco:eval_digit_map/1 and sends a sequence of events to it megaco:report_digit_event/2 in order to simplify testing of digit maps.

`encode_sdp(SDP) -> {ok, PP} | {error, Reason}`

Types:

```
SDP = sdp_property_parm() | sdp_property_group() | sdp_property_groups() |
asn1_NOVALUE
PP = property_parm() | property_group() | property_groups() | asn1_NOVALUE
Reason = term()
```

Encode (generate) an SDP construct.

If a `property_parm()` is found as part of the input (`SDP`) then it is left unchanged.

This function performs the following transformation:

- sdp() -> property_parm()
- sdp_property_group() -> property_group()
- sdp_property_groups() -> property_groups()

`decode_sdp(PP) -> {ok, SDP} | {error, Reason}`

Types:

```
PP = property_parm() | property_group() | property_groups() | asn1_NOVALUE
SDP = sdp() | decode_sdp_property_group() | decode_sdp_property_groups() |
asn1_NOVALUE
decode_sdp() = sdp() | {property_parm(), DecodeError}
decode_sdp_property_group() = [decode_sdp()]
decode_sdp_property_groups() = [decode_sdp_property_group()]
DecodeError = term()
Reason = term()
```

Decode (parse) a property parameter construct.

When decoding `property_group()` or `property_groups()`, those property parameter constructs that cannot be decoded (either because of decode error or because they are unknown), will be returned as a two-tuple. The first element of which will be the (undecoded) property parameter and the other the actual reason. This means that the caller of this function has to expect not only sdp-records, but also this two-tuple construct.

This function performs the following transformation:

- property_parm() -> sdp()
- property_group() -> sdp_property_group()
- property_groups() -> sdp_property_groups()

`get_sdp_record_from_PropertGroup(Type, PG) -> [sdp()]`

Types:

`Type = v | c | m | o | a | b | t | r | z | k | s | i | u | e | p`
`PG = sdp_property_group()`
`Reason = term()`

Retreive all the sdp records of type `Type` from the property group `PG`.

```
versions1() -> {ok, VersionInfo} | {error, Reason}
versions2() -> {ok, Info} | {error, Reason}
```

Types:

`VersionInfo = [version_info()]`
`version_info() = term()`
`Reason = term()`

Utility functions used to retrieve some system and application info.

The difference between the two functions is in how they get the modules to check. `versions1` uses the app-file and `versions2` uses the function `application:get_key`.

```
print_version_info() -> void()
print_version_info(VersionInfo) -> void()
```

Types:

`VersionInfo = [version_info()]`
`version_info() = term()`

Utility function to produce a formated printout of the versions info generated by the `versions1` and `versions2` functions.

The function print_version_info/0 uses the result of function version1/0 as `VersionInfo`.

Example:

```
        {ok, V} = megaco:versions1(), megaco:format_versions(V).
```

`enable_trace(Level, Destination) -> void()`

Types:

`Level = max | min | 0 <= integer() <= 100`
`Destination = File | Port | HandlerSpec | io`
`File = string()`
`Port = integer()`
`HandleSpec = {HandlerFun, Data}`
`HandleFun = fun() (two arguments)`

Data = term()

This function is used to start megaco tracing at a given `Level` and direct result to the given `Destination`.

It starts a tracer server and then sets the proper match spec (according to `Level`).

In the case when `Destination` is `File`, the printable megaco trace events will be printed to the file `File` using plain `io:format/2`.

In the case when `Destination` is `io`, the printable megaco trace events will be printed on stdout using plain `io:format/2`.

See `dbg` for further information.

```
disable_trace() -> void()
```

This function is used to stop megaco tracing.

```
set_trace(Level) -> void()
```

Types:

Level = max | min | 0 <= integer() <= 100

This function is used to change the megaco trace level.

It is assumed that tracing has already been enabled (see `enable_trace` above).

```
get_stats() -> {ok, TotalStats} | {error, Reason}
get_stats(GlobalCounter) -> {ok, CounterStats} | {error, Reason}
get_stats(ConnHandle) -> {ok, ConnHandleStats} | {error, Reason}
get_stats(ConnHandle, Counter) -> {ok, integer()} | {error, Reason}
```

Types:

TotalStats = [total_stats()]
total_stats() = {conn_handle(), [stats()]} | {global_counter(), integer()}
GlobalCounter = global_counter()
GlobalCounterStats = integer()
ConnHandle = conn_handle()
ConnHandleStats = [stats()]
stats() = {counter(), integer()}
Counter = counter()
counter() = medGwyGatewayNumTimerRecovery | medGwyGatewayNumErrors
global_counter() = medGwyGatewayNumErrors
Reason = term()

Retreive the (SNMP) statistic counters maintained by the megaco application. The global counters handle events that cannot be attributed to a single connection (e.g. protocol errors that occur before the connection has been properly setup).

```
reset_stats() -> void()
reset_stats(ConnHandle) -> void()
```

Types:

ConnHandle = conn_handle()

Reset all related (SNMP) statistics counters.

`test_request(ConnHandle, Version, EncodingMod, EncodingConfig, Actions) -> {MegaMsg, EncodeRes}`

Types:

ConnHandle = conn_handle()
Version = integer()
EncodingMod = atom()
EncodingConfig = Encoding configuration
Actions = A list
MegaMsg = #'MegacoMessage'{}
EncodeRes = {ok, Bin} | {error, Reason}
Bin = binary()
Reason = term()

Tests if the Actions argument is correctly composed.

This function is only intended for testing purposes. It's supposed to have a same kind of interface as the *call* or *cast* functions (with the additions of the `EncodingMod` and `EncodingConfig` arguments). It composes a complete megaco message end attempts to encode it. The return value, will be a tuple of the composed megaco message and the encode result.

`test_reply(ConnHandle, Version, EncodingMod, EncodingConfig, Reply) -> {MegaMsg, EncodeRes}`

Types:

ConnHandle = conn_handle()
Version = integer()
EncodingMod = atom()
EncodingConfig = A list
Reply = actual_reply()
MegaMsg = #'MegacoMessage'{}
EncodeRes = {ok, Bin} | {error, Reason}
Bin = binary()
Reason = term()

Tests if the Reply argument is correctly composed.

This function is only intended for testing purposes. It's supposed to test the `actual_reply()` return value of the callback functions *handle_trans_request* and *handle_trans_long_request* functions (with the additions of the `EncodingMod` and `EncodingConfig` arguments). It composes a complete megaco message end attempts to encode it. The return value, will be a tuple of the composed megaco message and the encode result.

megaco_edist_compress

Erlang module

The following functions should be exported from a `megaco_edist_compress` callback module:

Exports

`Module:encode(R, Version) -> T`

Types:

`R = megaco_encoder:megaco_message() | megaco_encoder:transaction() | megaco_encoder:action_reply() | megaco_encoder:action_request() | megaco_encoder:command_request()`

`Version = megaco_encoder:protocol_version()`

`T = term()`

Compress a megaco component. The erlang dist encoder makes no assumption on the how or even if the component is compressed.

`Module:decode(T, Version) -> R`

Types:

`T = term()`

`Version = megaco_encoder:protocol_version()`

`R = megaco_encoder:megaco_message() | megaco_encoder:transaction() | megaco_encoder:action_reply() | megaco_encoder:action_request() | megaco_encoder:command_request()`

Decompress a megaco component.

megaco_encoder

Erlang module

The following functions should be exported from a `megaco_encoder` callback module:

DATA TYPES

Note:

Note that the actual definition of (some of) these records depend on the megaco protocol version used. For instance, the `'TransactionReply'` record has two more fields in version 3, so a simple erlang type definition cannot be made here.

```
protocol_version() = integer()
segment_no()       = integer()
megaco_message() = #'MegacoMessage{}'
transaction() = {transactionRequest,     transaction_request()}      |
                {transactionPending,     transaction_reply()}        |
                {transactionReply,       transaction_pending()}      |
                {transactionResponseAck, transaction_response_ack()} |
                {segmentReply,           segment_reply()}
transaction_request() = #'TransactionRequest'{}
transaction_pending() = #'TransactionPending'{}
transaction_reply() = #'TransactionReply'{}
transaction_response_ack() = [transaction_ack()]
transaction_ack() = #'TransactionAck'{}
segment_reply() = #'SegmentReply'{}
action_request() = #'ActionRequest'{}
action_reply() = #'ActionReply'{}
command_request() = #'CommandRequest'{}
error_desc()   = #'ErrorDescriptor'{}
```

Exports

`Module:encode_message(EncodingConfig, Version, Message) -> {ok, Bin} | Error`

Types:

`EncodingConfig = list()`
`Version = integer()`
`Message = megaco_message()`
`Bin = binary()`
`Error = term()`

Encode a megaco message.

`Module:decode_message(EncodingConfig, Version, Bin) -> {ok, Message} | Error`

Types:

`EncodingConfig = list()`
`Version = integer() | dynamic`
`Message = megaco_message()`
`Bin = binary()`

Error = term()

Decode a megaco message.

Note that if the Version argument is `dynamic`, the decoder should try to figure out the actual version from the message itself and then use the proper decoder, e.g. version 1.
If on the other hand the Version argument is an integer, it means that this is the expected version of the message and the decoder for that version should be used.

`Module:decode_mini_message(EncodingConfig, Version, Bin) -> {ok, Message} | Error`

Types:

EncodingConfig = list()
Version = integer() | dynamic
Message = megaco_message()
Bin = binary()
Error = term()

Perform a minimal decode of a megaco message.

The purpose of this function is to do a minimal decode of Megaco message. A successfull result is a `'MegacoMessage'` in which only version and mid has been initiated. This function is used by the megaco_messenger module when the `decode_message/3` function fails to figure out the mid (the actual sender) of the message.

Note again that a successfull decode only returns a partially initiated message.

`Module:encode_transaction(EncodingConfig, Version, Transaction) -> OK | Error`

Types:

EncodingConfig = list()
Version = integer()
Transaction = transaction()
OK = {ok, Bin}
Bin = binary()
Error = {error, Reason}
Reason = not_implemented | OtherReason
OtherReason = term()

Encode a megaco transaction. If this, for whatever reason, is not supported, the function should return the error reason `not_implemented`.

This functionality is used both when the transaction sender is used and for segmentation. So, for either of those to work, this function **must** be fully supported!

`Module:encode_action_requests(EncodingConfig, Version, ARs) -> OK | Error`

Types:

EncodingConfig = list()
Version = integer()
ARs = action_requests()
action_requests() = [action_request()]
OK = {ok, Bin}

```
Bin = binary()
Error = {error, Reason}
Reason = not_implemented | OtherReason
OtherReason = term()
```

Encode megaco action requests. This function is called when the user calls the function *encode_actions/3*. If that function is never used or if the codec cannot support this (the encoding of individual actions), then return with error reason `not_implemented`.

`Module:encode_action_reply(EncodingConfig, Version, AR) -> OK | Error`

Types:

```
EncodingConfig = list()
Version = integer()
AR = action_reply()
OK = {ok, Bin}
Bin = binary()
Error = {error, Reason}
Reason = not_implemented | OtherReason
OtherReason = term()
```

Encode a megaco action reply. If this, for whatever reason, is not supported, the function should return the error reason `not_implemented`.

This function is used when segmentation has been configured. So, for this to work, this function **must** be fully supported!

megaco_transport

Erlang module

The following functions should be exported from a `megaco_transport` callback module:

- *send_message/2* [`mandatory`]
- *send_message/3* [`optional`]
- *resend_message/2* [`optional`]
-

Exports

```
Module:send_message(Handle, Msg) -> ok | {cancel, Reason} | Error
Module:send_message(Handle, Msg, Resend) -> ok | {cancel, Reason} | Error
```

Types:

```
Handle = term()
Msg = binary() | iolist()
Resend = boolean()
Reason = term()
Error = term()
```

Send a megaco message.

If the function returns `{cancel, Reason}`, this means the transport module decided not to send the message. This is **not** an error. No error messages will be issued and no error counters incremented. What actions this will result in depends on what kind of message was sent.

In the case of requests, megaco will cancel the message in much the same way as if `megaco:cancel` had been called (after a successfull send). The information will be propagated back to the user differently depending on how the request(s) where issued: For requests issued using *megaco:call*, the info will be delivered in the return value. For requests issued using `megaco:cast` the info will be delivered via a call to the callback function *handle_trans_reply*.

In the case of reply, megaco will cancel the reply and information of this will be returned to the user via a call to the callback function *handle_trans_ack*.

The function `send_message/3` will only be called if the *resend_indication* config option has been set to the value `flag`. The third argument, `Resend` then indicates if the message send is a resend or not.

```
Module:resend_message(Handle, Msg) -> ok | {cancel, Reason} | Error
```

Types:

```
Handle = term()
Msg = binary() | iolist()
Reason = term()
Error = term()
```

Re-send a megaco message.

Note that this function will only be called if the user has set the *resend_indication* config option to `true`**and** it is in fact a message resend. If not **both** of these condition's are meet, `send_message` will be called.

If the function returns `{cancel, Reason}`, this means the transport module decided not to send the message. This is **not** an error. No error messages will be issued and no error counters incremented. What actions this will result in depends on what kind of message was sent.

In the case of requests, megaco will cancel the message in much the same way as if `megaco:cancel` had been called (after a successfull send). The information will be propagated back to the user differently depending on how the request(s) where issued: For requests issued using *megaco:call*, the info will be delivered in the return value. For requests issued using `megaco:cast` the info will be delivered via a call to the callback function *handle_trans_reply*.

In the case of reply, megaco will cancel the reply and information of this will be returned to the user via a call to the callback function *handle_trans_ack*.

megaco_tcp

Erlang module

This module contains the public interface to the TPKT (TCP/IP) version transport protocol for Megaco/H.248.

Exports

`start_transport() -> {ok, TransportRef}`

Types:

TransportRef = pid()

This function is used for starting the TCP/IP transport service. Use exit(TransportRef, Reason) to stop the transport service.

`listen(TransportRef, ListenPortSpecList) -> ok`

Types:

TransportRef = pid() | regname()
OptionListPerPort = [Option]
Option = {port, integer()} |{options, list()} |{receive_handle, term()}

This function is used for starting new TPKT listening socket for TCP/IP. The option list contains the socket definitions.

`connect(TransportRef, OptionList) -> {ok, Handle, ControlPid} | {error, Reason}`

Types:

TransportRef = pid() | regname()
OptionList = [Option]
Option = {host, Ipaddr} | {port, integer()} |{options, list()} | {receive_handle, term()} |{module, atom()}
Handle = socket_handle()
ControlPid = pid()
Reason = term()

This function is used to open a TPKT connection.

The `module` option makes it possible for the user to provide their own callback module. The `receive_message/4` or `process_received_message/4` functions of this module is called when a new message is received (which one depends on the size of the message; small - receive_message, large - process_received_message). Default value is **megaco**.

`close(Handle) -> ok`

Types:

Handle = socket_handle()

This function is used for closing an active TPKT connection.

`socket(Handle) -> Socket`

Types:

Handle = socket_handle()
Socket = inet_socket()

This function is used to convert a socket_handle() to a inet_socket(). inet_socket() is a plain socket, see the inet module for more info.

```
send_message(Handle, Message) -> ok
```

Types:

Handle = socket_handle()
Message = binary() | iolist()

Sends a message on a connection.

```
block(Handle) -> ok
```

Types:

Handle = socket_handle()

Stop receiving incoming messages on the socket.

```
unblock(Handle) -> ok
```

Types:

Handle = socket_handle()

Starting to receive incoming messages from the socket again.

```
upgrade_receive_handle(ControlPid) -> ok
```

Types:

ControlPid = pid()

Update the receive handle of the control process (e.g. after having changed protocol version).

```
get_stats() -> {ok, TotalStats} | {error, Reason}
get_stats(SendHandle) -> {ok, SendHandleStats} | {error, Reason}
get_stats(SendHandle, Counter) -> {ok, CounterStats} | {error, Reason}
```

Types:

TotalStats = [send_handle_stats()]
total_stats() = {send_handle(), [stats()]}
SendHandle = send_handle()
SendHandleStats = [stats()]
Counter = tcp_stats_counter()
CounterStats = integer()
stats() = {tcp_stats_counter(), integer()}
tcp_stats_counter() = medGwyGatewayNumInMessages | medGwyGatewayNumInOctets | medGwyGatewayNumOutMessages | medGwyGatewayNumOutOctets | medGwyGatewayNumErrors
Reason = term()

Retreive the TCP related (SNMP) statistics counters.

```
reset_stats() -> void()
reset_stats(SendHandle) -> void()
```

Types:

`SendHandle = send_handle()`

Reset all TCP related (SNMP) statistics counters.

megaco_udp

Erlang module

This module contains the public interface to the UDP/IP version transport protocol for Megaco/H.248.

Exports

`start_transport() -> {ok, TransportRef}`

Types:

TransportRef = pid()

This function is used for starting the UDP/IP transport service. Use exit(TransportRef, Reason) to stop the transport service.

`open(TransportRef, OptionList) -> {ok, Handle, ControlPid} | {error, Reason}`

Types:

TransportRef = pid() | regname()
OptionList = [option()]
option() = {port, integer()} |{options, list()} |{receive_handle, receive_handle()} |{module, atom()}
Handle = socket_handle()
receive_handle() = term()
ControlPid = pid()
Reason = term()

This function is used to open an UDP/IP socket.

The `module` option makes it possible for the user to provide their own callback module. The functions `receive_message/4` or `process_received_message/4` of this module is called when a new message is received (which one depends on the size of the message; small - receive_message, large - process_received_message). Default value is **megaco**.

`close(Handle, Msg) -> ok`

Types:

Handle = socket_handle()
Msg

This function is used for closing an active UDP socket.

`socket(Handle) -> Socket`

Types:

Handle = socket_handle()
Socket = inet_socket()

This function is used to convert a socket_handle() to a inet_socket(). inet_socket() is a plain socket, see the inet module for more info.

```
create_send_handle(Handle, Host, Port) -> send_handle()
```

Types:

Handle = socket_handle()
Host = {A,B,C,D} | string()
Port = integer()

Creates a send handle from a transport handle. The send handle is intended to be used by megaco_udp:send_message/2.

```
send_message(SendHandle, Msg) -> ok
```

Types:

SendHandle = send_handle()
Message = binary() | iolist()

Sends a message on a socket. The send handle is obtained by megaco_udp:create_send_handle/3. Increments the NumOutMessages and NumOutOctets counters if message successfully sent. In case of a failure to send, the NumErrors counter is **not** incremented. This is done elsewhere in the megaco app.

```
block(Handle) -> ok
```

Types:

Handle = socket_handle()

Stop receiving incoming messages on the socket.

```
unblock(Handle) -> ok
```

Types:

Handle = socket_handle()

Starting to receive incoming messages from the socket again.

```
upgrade_receive_handle(ControlPid, NewHandle) -> ok
```

Types:

ControlPid = pid()
NewHandle = receive_handle()
receive_handle() = term()

Update the receive handle of the control process (e.g. after having changed protocol version).

```
get_stats() -> {ok, TotalStats} | {error, Reason}
get_stats(SendHandle) -> {ok, SendHandleStats} | {error, Reason}
get_stats(SendHandle, Counter) -> {ok, CounterStats} | {error, Reason}
```

Types:

TotalStats = [total_stats()]
total_stats() = {send_handle(), [stats()]}
SendHandle = send_handle()
SendHandleStats = [stats()]
Counter = udp_stats_counter()
CounterStats = integer()
stats() = {udp_stats_counter(), integer()}

tcp_stats_counter() = medGwyGatewayNumInMessages | medGwyGatewayNumInOctets | medGwyGatewayNumOutMessages | medGwyGatewayNumOutOctets | medGwyGatewayNumErrors

Reason = term()

Retreive the UDP related (SNMP) statistics counters.

```
reset_stats() -> void()
reset_stats(SendHandle) -> void()
```

Types:

SendHandle = send_handle()

Reset all TCP related (SNMP) statistics counters.

megaco_user

Erlang module

This module defines the callback behaviour of Megaco users. A megaco_user compliant callback module must export the following functions:

- *handle_connect/2,3*
- *handle_disconnect/3*
- *handle_syntax_error/3,4*
- *handle_message_error/3,4*
- *handle_trans_request/3,4*
- *handle_trans_long_request/3,4*
- *handle_trans_reply/4,5*
- *handle_trans_ack/4,5*
- *handle_unexpected_trans/3,4*
- *handle_trans_request_abort/4,5*
- *handle_segment_reply/5,6*

The semantics of them and their exact signatures are explained below.

The `user_args` configuration parameter which may be used to extend the argument list of the callback functions. For example, the handle_connect function takes by default two arguments:

```
        handle_connect(Handle, Version)
```

but if the `user_args` parameter is set to a longer list, such as `[SomePid,SomeTableRef]`, the callback function is expected to have these (in this case two) extra arguments last in the argument list:

```
        handle_connect(Handle, Version, SomePid, SomeTableRef)
```

Note:

Must of the functions below has an optional `Extra` argument (e.g. *handle_unexpected_trans/4*). The functions which takes this argument will be called if and only if one of the functions *receive_message/5* or *process_received_message/5* was called with the `Extra` argument different than `ignore_extra`.

DATA TYPES

```
action_request() = #'ActionRequest'{}
action_reply() = #'ActionReply'{}
error_desc() = #'ErrorDescriptor'{}
segment_no() = integer()
```

```
conn_handle() = #megaco_conn_handle{}
```

The record initially returned by `megaco:connect/4,5`. It identifies a "virtual" connection and may be reused after a reconnect (disconnect + connect).

```
protocol_version() = integer()
```

Is the actual protocol version. In most cases the protocol version is retrieved from the processed message, but there are exceptions:

- When `handle_connect/2,3` is triggered by an explicit call to `megaco:connect/4,5`.
- `handle_disconnect/3`
- `handle_syntax_error/3`

In these cases, the ProtocolVersion default version is obtained from the static connection configuration:

- `megaco:conn_info(ConnHandle, protocol_version)`.

Exports

```
handle_connect(ConnHandle, ProtocolVersion) -> ok | error |
{error,ErrorDescr}
handle_connect(ConnHandle, ProtocolVersion, Extra) -> ok | error |
{error,ErrorDescr}
```

Types:

`ConnHandle = conn_handle()`
`ProtocolVersion = protocol_version()`
`ErrorDescr = error_desc()`
`Extra = term()`

Invoked when a new connection is established

Connections may either be established by an explicit call to megaco:connect/4 or implicitly at the first invocation of megaco:receive_message/3.

Normally a Media Gateway (MG) connects explicitly while a Media Gateway Controller (MGC) connects implicitly.

At the Media Gateway Controller (MGC) side it is possible to reject a connection request (and send a message error reply to the gateway) by returning `{error, ErrorDescr}` or simply `error` which generates an error descriptor with code 402 (unauthorized) and reason "Connection refused by user" (this is also the case for all unknown results, such as exit signals or throw).

See *note* above about the `Extra` argument in `handle_message_error/4`.

`handle_connect/3` (with `Extra`) can also be called as a result of a call to the *megaco:connect/5* function (if that function is called with the `Extra` argument different than `ignore_extra`.

```
handle_disconnect(ConnHandle, ProtocolVersion, Reason) -> ok
```

Types:

`ConnHandle = conn_handle()`
`ProtocolVersion = protocol_version()`
`Reason = term()`

Invoked when a connection is teared down

The disconnect may either be made explicitly by a call to megaco:disconnect/2 or implicitly when the control process of the connection dies.

```
handle_syntax_error(ReceiveHandle, ProtocolVersion, DefaultED) -> reply |
{reply, ED} | no_reply | {no_reply, ED}
handle_syntax_error(ReceiveHandle, ProtocolVersion, DefaultED, Extra) ->
reply | {reply, ED} | no_reply | {no_reply, ED}
```

Types:

ReceiveHandle = receive_handle()
ProtocolVersion = protocol_version()
DefaultED = error_desc()
ED = error_desc()
Extra = term()

Invoked when a received message had syntax errors

Incoming messages is delivered by megaco:receive_message/4 and normally decoded successfully. But if the decoding failed this function is called in order to decide if the originator should get a reply message (reply) or if the reply silently should be discarded (no_reply).

Syntax errors are detected locally on this side of the protocol and may have many causes, e.g. a malfunctioning transport layer, wrong encoder/decoder selected, bad configuration of the selected encoder/decoder etc.

The error descriptor defaults to `DefaultED`, but can be overridden with an alternate one by returning `{reply,ED}` or `{no_reply,ED}` instead of `reply` and `no_reply` respectively.

Any other return values (including exit signals or throw) and the `DefaultED` will be used.

See *note* above about the `Extra` argument in `handle_syntax_error/4`.

```
handle_message_error(ConnHandle, ProtocolVersion, ErrorDescr) -> ok
handle_message_error(ConnHandle, ProtocolVersion, ErrorDescr, Extra) -> ok
```

Types:

ConnHandle = conn_handle()
ProtocolVersion = protocol_version()
ErrorDescr = error_desc()
Extra = term()

Invoked when a received message just contains an error instead of a list of transactions.

Incoming messages is delivered by megaco:receive_message/4 and successfully decoded. Normally a message contains a list of transactions, but it may instead contain an ErrorDescriptor on top level of the message.

Message errors are detected remotely on the other side of the protocol. And you probably don't want to reply to it, but it may indicate that you have outstanding transactions that not will get any response (request -> reply; reply -> ack).

See *note* above about the `Extra` argument in `handle_message_error/4`.

```
handle_trans_request(ConnHandle, ProtocolVersion, ActionRequests) ->
pending() | reply() | ignore_trans_request
handle_trans_request(ConnHandle, ProtocolVersion, ActionRequests, Extra) ->
pending() | reply() | ignore_trans_request
```

Types:

ConnHandle = conn_handle()
ProtocolVersion = protocol_version()
ActionRequests = [action_request()]
Extra = term()
pending() = {pending, req_data()}
req_data() = term()
reply() = {ack_action(), actual_reply()} | {ack_action(), actual_reply(), send_options()}

```
ack_action() = discard_ack | {handle_ack, ack_data()} |
{handle_pending_ack, ack_data()} | {handle_sloppy_ack, ack_data()}
actual_reply() = [action_reply()] | error_desc()
ack_data() = term()
send_options() = [send_option()]
send_option() = {reply_timer, megaco_timer()} | {send_handle, term()} |
{protocol_version, integer()}
Extra = term()
```

Invoked for each transaction request

Incoming messages is delivered by megaco:receive_message/4 and successfully decoded. Normally a message contains a list of transactions and this function is invoked for each TransactionRequest in the message.

This function takes a list of 'ActionRequest' records and has three main options:

`Return ignore_trans_request`

Decide that these action requests shall be ignored completely.

`Return pending()`

Decide that the processing of these action requests will take a long time and that the originator should get an immediate 'TransactionPending' reply as interim response. The actual processing of these action requests instead should be delegated to the the handle_trans_long_request/3 callback function with the req_data() as one of its arguments.

`Return reply()`

Process the action requests and either return an error_descr() indicating some fatal error or a list of action replies (wildcarded or not).

If for some reason megaco is unable to deliver the reply, the reason for this will be passed to the user via a call to the callback function *handle_trans_ack*, unless `ack_action() = discard_ack`.

The ack_action() is either:

`discard_ack`

Meaning that you don't care if the reply is acknowledged or not.

`{handle_ack, ack_data()} | {handle_ack, ack_data(), send_options()}`

Meaning that you want an immediate acknowledgement when the other part receives this transaction reply. When the acknowledgement eventually is received, the handle_trans_ack/4 callback function will be invoked with the ack_data() as one of its arguments. ack_data() may be any Erlang term.

`{handle_pending_ack, ack_data()} | {handle_pending_ack, ack_data(), send_options()}`

This has the same effect as the above, **if and only if** megaco has sent at least one pending message for this request (during the processing of the request). If no pending message has been sent, then immediate acknowledgement will **not** be requested.

Note that this only works as specified if the `sent_pending_limit` config option has been set to an integer value.

`{handle_sloppy_ack, ack_data()}| {handle_sloppy_ack, ack_data(), send_options()}`

Meaning that you want an acknowledgement **sometime**. When the acknowledgement eventually is received, the handle_trans_ack/4 callback function will be invoked with the ack_data() as one of its arguments. ack_data() may be any Erlang term.

Any other return values (including exit signals or throw) will result in an error descriptor with code 500 (internal gateway error) and the module name (of the callback module) as reason.

See *note* above about the `Extra` argument in `handle_trans_request/4`.

```
handle_trans_long_request(ConnHandle, ProtocolVersion, ReqData) -> reply()
handle_trans_long_request(ConnHandle, ProtocolVersion, ReqData, Extra) ->
reply()
```

Types:

```
ConnHandle = conn_handle()
ProtocolVersion = protocol_version()
ReqData = req_data()
Extra = term()
req_data() = term()
reply() = {ack_action(), actual_reply()} | {ack_action(), actual_reply(),
send_options()}
ack_action() = discard_ack | {handle_ack, ack_data()} |
{handle_sloppy_ack, ack_data()}
actual_reply() = [action_reply()] | error_desc()
ack_data() = term()
send_options() = [send_option()]
send_option() = {reply_timer, megaco_timer()} | {send_handle, term()} |
{protocol_version, integer()}
Extra = term()
```

Optionally invoked for a time consuming transaction request

If this function gets invoked or not is controlled by the reply from the preceding call to handle_trans_request/3. The handle_trans_request/3 function may decide to process the action requests itself or to delegate the processing to this function.

The req_data() argument to this function is the Erlang term returned by handle_trans_request/3.

Any other return values (including exit signals or throw) will result in an error descriptor with code 500 (internal gateway error) and the module name (of the callback module) as reason.

See *note* above about the `Extra` argument in `handle_trans_long_request/4`.

```
handle_trans_reply(ConnHandle, ProtocolVersion, UserReply, ReplyData) -> ok
handle_trans_reply(ConnHandle, ProtocolVersion, UserReply, ReplyData, Extra)
-> ok
```

Types:

```
ConnHandle = conn_handle()
ProtocolVersion = protocol_version()
UserReply = success() | failure()
success() = {ok, result()}
result() = transaction_result() | segment_result()
transaction_result() = action_reps()
segment_result() = {segment_no(), last_segment(), action_reps()}
action_reps() = [action_reply()]
```

```
failure() = {error, reason()} | {error, ReplyNo, reason()}
reason() = transaction_reason() | segment_reason() | user_cancel_reason()
| send_reason() | other_reason()
transaction_reason() = error_desc()
segment_reason() = {segment_no(), last_segment(), error_desc()}
other_reason() = timeout | {segment_timeout, missing_segments()} |
exceeded_recv_pending_limit | term()
last_segment() = bool()
missing_segments() = [segment_no()]
user_cancel_reason() = {user_cancel, reason_for_user_cancel()}
reason_for_user_cancel() = term()
send_reason() = send_cancelled_reason() | send_failed_reason()
send_cancelled_reason() = {send_message_cancelled,
reason_for_send_cancel()}
reason_for_send_cancel() = term()
send_failed_reason() = {send_message_failed, reason_for_send_failure()}
reason_for_send_failure() = term()
ReplyData = reply_data()
ReplyNo = integer() > 0
reply_data() = term()
Extra = term()
```

Optionally invoked for a transaction reply

The sender of a transaction request has the option of deciding, whether the originating Erlang process should synchronously wait (`megaco:call/3`) for a reply or if the message should be sent asynchronously (`megaco:cast/3`) and the processing of the reply should be delegated this callback function.

Note that if the reply is segmented (split into several smaller messages; segments), then some extra info, segment number and an indication if all segments of a reply has been received or not, is also included in the `UserReply`.

The `ReplyData` defaults to `megaco:lookup(ConnHandle, reply_data)`, but may be explicitly overridden by a `megaco:cast/3` option in order to forward info about the calling context of the originating process.

At `success()`, the `UserReply` either contains:

- A list of 'ActionReply' records possibly containing error indications.
- A tuple of size three containing: the segment number, the `last segment indicator` and finally a list of 'ActionReply' records possibly containing error indications. This is of course only possible if the reply was segmented.

`failure()` indicates an local or external error and can be one of the following:

- A `transaction_reason()`, indicates that the remote user has replied with an explicit transactionError.
- A `segment_reason()`, indicates that the remote user has replied with an explicit transactionError for this segment. This is of course only possible if the reply was segmented.
- A `user_cancel_reason()`, indicates that the request has been canceled by the user. `reason_for_user_cancel()` is the reason given in the call to the *cancel* function.

- A `send_reason()`, indicates that the transport module *send_message* function did not send the message. The reason for this can be:
  - `send_cancelled_reason()` - the message sending was deliberately cancelled. `reason_for_send_cancel()` is the reason given in the `cancel` return from the *send_message* function.
  - `send_failed_reason()` - an error occurred while attempting to send the message.
- An `other_reason()`, indicates some other error such as:
  - `timeout` - the reply failed to arrive before the request timer expired.
  - `{segment_timeout, missing_segments()}` - one or more segments was not delivered before the expire of the segment timer.
  - `exceeded_recv_pending_limit` - the pending limit was exceeded for this request.

See *note* above about the `Extra` argument in `handle_trans_reply/5`.

```
handle_trans_ack(ConnHandle, ProtocolVersion, AckStatus, AckData) -> ok
handle_trans_ack(ConnHandle, ProtocolVersion, AckStatus, AckData, Extra) ->
ok
```

Types:

```
ConnHandle = conn_handle()
ProtocolVersion = protocol_version()
AckStatus = ok | {error, reason()}
reason() = user_cancel_reason() | send_reason() | other_reason()
user_cancel_reason() = {user_cancel, reason_for_user_cancel()}
send_reason() = send_cancelled_reason() | send_failed_reason()
send_cancelled_reason() = {send_message_cancelled,
reason_for_send_cancel()}
reason_for_send_cancel() = term()
send_failed_reason() = {send_message_failed, reason_for_send_failure()}
reason_for_send_failure() = term()
other_reason() = term()
AckData = ack_data()
ack_data() = term()
Extra = term()
```

Optionally invoked for a transaction acknowledgement

If this function gets invoked or not, is controlled by the reply from the preceding call to handle_trans_request/3. The handle_trans_request/3 function may decide to return {handle_ack, ack_data()} or {handle_sloppy_ack, ack_data()} meaning that you need an immediate acknowledgement of the reply and that this function should be invoked to handle the acknowledgement.

The ack_data() argument to this function is the Erlang term returned by handle_trans_request/3.

If the AckStatus is ok, it is indicating that this is a true acknowledgement of the transaction reply.

If the AckStatus is {error, Reason}, it is an indication that the acknowledgement or even the reply (for which this is an acknowledgement) was not delivered, but there is no point in waiting any longer for it to arrive. This happens when:

`reply_timer`

The `reply_timer` eventually times out.

reply send failure

When megaco fails to send the reply (see *handle_trans_reply*), for whatever reason.

cancel

The user has explicitly cancelled the wait (megaco:cancel/2).

See *note* above about the `Extra` argument in `handle_trans_ack/5`.

```
handle_unexpected_trans(ConnHandle, ProtocolVersion, Trans) -> ok
handle_unexpected_trans(ConnHandle, ProtocolVersion, Trans, Extra) -> ok
```

Types:

```
ConnHandle = conn_handle()
ProtocolVersion = protocol_version()
Trans = #'TransactionPending'{} | #'TransactionReply'{} |
#'TransactionResponseAck'{}
Extra = term()
```

Invoked when a unexpected message is received

If a reply to a request is not received in time, the megaco stack removes all info about the request from its tables. If a reply should arrive after this has been done the app has no way of knowing where to send this message. The message is delivered to the "user" by calling this function on the local node (the node which has the link).

See *note* above about the `Extra` argument in `handle_unexpected_trans/4`.

```
handle_trans_request_abort(ConnHandle, ProtocolVersion, TransNo, Pid) -> ok
handle_trans_request_abort(ConnHandle, ProtocolVersion, TransNo, Pid, Extra)
-> ok
```

Types:

```
ConnHandle = conn_handle()
ProtocolVersion = protocol_version()
TransNo = integer()
Pid = undefined | pid()
Extra = term()
```

Invoked when a transaction request has been aborted

This function is invoked if the originating pending limit has been exceeded. This usually means that a request has taken abnormally long time to complete.

See *note* above about the `Extra` argument in `handle_trans_request_abort/5`.

```
handle_segment_reply(ConnHandle, ProtocolVersion, TransNo, SegNo, SegCompl) -
> ok
handle_segment_reply(ConnHandle, ProtocolVersion, TransNo, SegNo, SegCompl,
Extra) -> ok
```

Types:

```
ConnHandle = conn_handle()
ProtocolVersion = protocol_version()
TransNo = integer()
SegNo = integer()
```

```
SegCompl = asn1_NOVALUE | 'NULL'
Extra = term()
```

This function is called when a segment reply has been received if the *segment_reply_ind* config option has been set to true.

This is in effect a progress report.

See *note* above about the `Extra` argument in `handle_segment_reply/6`.

megaco_flex_scanner

Erlang module

This module contains the public interface to the flex scanner linked in driver. The flex scanner performs the scanning phase of text message decoding.

The flex scanner is written using a tool called **flex**. In order to be able to compile the flex scanner driver, this tool has to be available.

By default the flex scanner reports line-number of an error. But it can be built without line-number reporting. Instead token number is used. This will speed up the scanning some 5-10%. Use `--disable-megaco-flex-scanner-lineno` when configuring the application.

The scanner will, by default, be built as a reentrant scanner **if** the flex utility supports this (it depends on the version of flex). It is possible to explicitly disable this even when flex support this. Use `--disable-megaco-reentrant-flex-scanner` when configuring the application.

DATA TYPES

```
megaco_ports() = term()
megaco_version() = integer() >= 1
```

Exports

`start() -> {ok, PortOrPorts} | {error, Reason}`

Types:

`PortOrPorts = megaco_ports()`

`Reason = term()`

This function is used to start the flex scanner. It locates the library and loads the linked in driver.

On a single core system or if it's a non-reentrant scanner, a single port is created. On a multi-core system with a reentrant scanner, several ports will be created (one for each scheduler).

Note that the process that calls this function **must** be permanent. If it dies, the port(s) will exit and the driver unload.

`stop(PortOrPorts) -> stopped`

Types:

`PortOrPorts = megaco_ports()`

This function is used to stop the flex scanner. It also unloads the driver.

`is_reentrant_enabled() -> Boolean`

Types:

`Boolean = boolean()`

Is the flex scanner reentrant or not.

`is_scanner_port(Port, PortOrPorts) -> Boolean`

Types:

`Port = port()`

PortOrPorts = megaco_ports()
Boolean = boolean()

Checks if a port is a flex scanner port or not (useful when if a port exits).

```
scan(Binary, PortOrPorts) -> {ok, Tokens, Version, LatestLine} | {error,
Reason, LatestLine}
```

Types:

Binary = binary()
PortOrPorts = megaco_ports()
Tokens = list()
Version = megaco_version()
LatestLine = integer()
Reason = term()

Scans a megaco message and generates a token list to be passed on the parser.

megaco_codec_meas

Erlang module

This module implements a simple megaco codec measurement tool.

Results are written to file (excel compatible text files) and on stdout.

Note that this module is **not** included in the runtime part of the application.

Exports

```
start() -> void()
start(MessagePackage) -> void()
```

Types:

```
MessagePackageRaw = message_package()
message_package() = atom()
```

This function runs the measurement on all the **official** codecs; pretty, compact, ber, per and erlang.

megaco_codec_mstone1

Erlang module

This module implements the **mstone1** tool, a simple megaco codec-based performance tool.

The results, the mstone value(s), are written to stdout.

Note that this module is **not** included in the runtime part of the application.

Exports

```
start() -> void()
start(MessagePackage) -> void()
start(MessagePackage, Factor) -> void()
```

Types:

MessagePackage = message_package()
message_package() = atom()
Factor() = integer() > 0

This function starts the **mstone1** performance test with all codec configs. `Factor` (defaults to `1`) processes are started for every supported codec config.

Each process encodes and decodes their messages. The number of messages processed in total (for all processes) is the mstone value.

```
start_flex() -> void()
start_flex(MessagePackage) -> void()
start_flex(MessagePackage, Factor) -> void()
```

Types:

MessagePackage = message_package()
message_package() = atom()
Factor() = integer() > 0

This function starts the **mstone1** performance test with only the flex codec configs (i.e. `pretty` and `compact` with `flex`). The same number of processes are started as when running the standard test (using the `start/0,1` function). Each process encodes and decodes their messages. The number of messages processed in total (for all processes) is the mstone value.

```
start_only_drv() -> void()
start_only_drv(MessagePackage) -> void()
start_only_drv(MessagePackage, Factor) -> void()
```

Types:

MessagePackage = message_package()
message_package() = atom()
Factor() = integer() > 0

This function starts the **mstone1** performance test with only the driver using codec configs (i.e. `pretty` and `compact` with `flex`, and `ber` and `per` with `driver` and `erlang` with `compressed`). The same number of processes are

started as when running the standard test (using the `start/0,1` function). Each process encodes and decodes their messages. The number of messages processed in total (for all processes) is the mstone value.

```
start_no_drv() -> void()
start_no_drv(MessagePackage) -> void()
start_no_drv(MessagePackage, Factor) -> void()
```

Types:

```
MessagePackage = message_package()
message_package() = atom()
Factor() = integer() > 0
```

This function starts the **mstone1** performance test with codec configs not using any drivers (i.e. `pretty` and `compact` without `flex`, `ber` and `per` without `driver` and `erlang` without `compressed`). The same number of processes are started as when running the standard test (using the `start/0,1` function). Each process encodes and decodes their messages. The number of messages processed in total (for all processes) is the mstone value.

megaco_codec_mstone2

Erlang module

This module implements the **mstone2** tool, a simple megaco codec-based performance tool.

The results, the mstone value(s), are written to stdout.

Note that this module is **not** included in the runtime part of the application.

Exports

```
start() -> void()
start(MessagePackage) -> void()
```

Types:

`MessagePackage = message_package()`

`message_package() = atom()`

This function starts the **mstone2** performance test with all codec configs. Processes are created dynamically. Each process make **one** run through their messages (decoding and encoding messages) and then exits. When one process exits, a new is created with the same codec config and set of messages.

The number of messages processed in total (for all processes) is the mstone value.

megaco_codec_transform

Erlang module

This module implements a simple megaco message transformation utility.

Note that this module is **not** included in the runtime part of the application.

Exports

```
export_messages() -> void()
export_messages(MessagePackage) -> void()
```

Types:

`MessagePackage = atom()`

Export the messages in the `MessagePackage` (default is `time_test`).

The output produced by this function is a directory structure with the following structure:

```
<message package>/pretty/<message-files>
                  compact/<message-files>
                  per/<message-files>
                  ber/<message-files>
                  erlang/<message-files>
```