Implementing H.323 terminal: H.245 Subsystem

Michał Konrad Rój

Implementing H.323 terminal: H.245 Subsystem

by Michał Konrad Rój

Revision History

Revision 0.8 25 Nov 2001
(After one year-break): corrections;
Revision 0.7 22 Sep 2000
Corrections; This version was submitted as the author's B.Sc. thesis
Revision 0.6 19 Sep 2000
Added: API proporsals; improved: entities description
Revision 0.5 15 Sep 2000
Many errors found; improved diagrams
Revision 0.4 5 Sep 2000
Added: sequence diagrams
Revision 0.3 1 Sep 2000
Added: tutorial, implementation proporsal; Improved: figures
Revision 0.2 23 Aug 2000
Added: entities description, figures
Revision 0.1 16 Aug 2000
First draft for supervisor's reviev

Abstract

This work relates to H.245 control protocol, the protocol that is used to exchange signalling information between H.323 endpoints. It is defined by ITU-T Recommendation H.245. Although all the descriptions is this work refer to the H.323 multimedia terminal, most of the H.245 capabilities refer also to the other endpoints (MCU, Gatekeepers, H-series terminals).

The first chapter introduces the essential terms and capabilities of a system based on ITU-T Rec. H.245. Since the compatibility with the Recommendation is the most important goal of this work, the chapter is elemental. In the next chapter all mandatory H.245 parts in H.323 terminal are described in details. Finally, a design and implementation of the H.245-based system are described. The system was developed during the author's gradutating process.

This work does *not* describe the ASN.1 encoding (but it has to be mentioned here) nor Call Signalling procedures used to establish the control channel. These aspects are beyond the scope of this work.

Table of Contents

List of Figures

Chapter 1. H.245 Tutorial

This tutorial introduces fundamental concepts of the H.245 communication protocol. The chapter is based on the recommendation [1] only; what follows is the author's interpretation of the standard. The way H.245 is used in the H.323 terminal is covered in Chapter 2.

1.1. Rationale

Almost every contemporary communication system requires one or more signalization channels apart from the channels where multimedia data is exchanged. For example, multimedia terminals must establish the exact moment of the beginning of the transmission to not to lose any data.

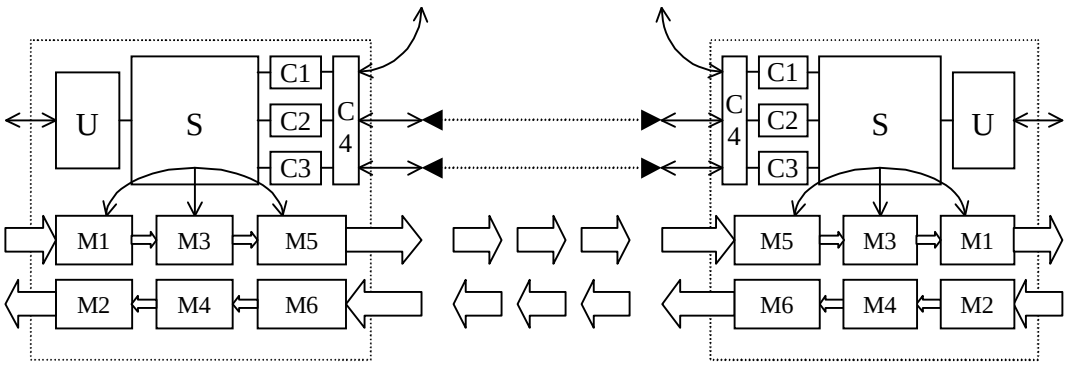

Figure 1-1. Connection between H.323 terminals

The typical scenario of H.323 terminal connection is outlined in Figure 1-1. Two terminals (dotted rectangles) have a special modules (signed as C1..C4), which make up the *System Control Unit* of the H.323 terminal. The first signalization (called *RAS*) is performed between the terminal and a so-called *Gatekeeper*. The RAS link is represented as a curved line (the top of the figure). Two next signalization links are established between presented terminals. They are represented as the two horizontal dotted lines. The upper link is a *call signalling* link, controlled by the *Call Signalling* module (C2). It is opened between two terminals to signal that a new connection is requested. If the call signalling procedures succeed (the called terminal agrees to open a new session), a so-called *control channel* is opened (the lower link). After that the call signalling connection is usually closed (though it can be used later for additional signalling purposes). The two lower links, which are illustrated by multiple block arrows at the figure, represent media transmission, which follows RTP/RTCP protocol.

All the mentioned signalization links have their appropriate functionality. They are defined by different recommendations, for instance RAS and call signalling are defined in *ITU-T Rec. H.225.0* [5]. The control channel is defined in *ITU-T Rec. H.245* [1], while RTP/RTCP is defined in *RFC 1889* [6]. The most complex signalization system is the one related to the control channel (described here). The following functions are performed over the control channel (the list is not exhaustive):

- Exchanging properties of terminals. It is used to inform the remote terminal about supported capabilities. The capabilities are supported audio/video codecs, protocol identifier, and modes a terminal can work in. Once the exchange is successfully finished, a terminal knows what types of audio/video formats the other end can understand.
- Opening and closing logical channels. Once a terminal knows about the other end's capabilities it can choose the format (encoding) of media it wants to transmit. It notifies the other of its choice by opening logical channels.
- Measuring delay in the control channel. Checking if the other endpoint is still alive.
- Establishing and managing maintenance loops.
- Picking the master terminal. A successful completion of some of the activities mentioned above requires designating one of the terminals as "master" and the other as "slave" (there is no democracy in multimedia terminals). Only the master terminal can initiate certain actions.
- Closing the session between the two endpoints.

These tasks cannot be performed without a special protocol. H.323 terminals use such a protocol, the one defined in *ITU-T Rec. H.245*. The recommendation defines a peer-to-peer protocol, which may be used by a number of different teleconferencing systems (e.g., based on ITU-T Rec. H.310, H.323, H.324 [9]). Not all of the H.245 features are used by any of these systems. The H.323 terminal uses just a subset of all the H.245 features.

1.2. Basic Concepts

1.2.1. Control Channel

After the termination of call signalling procedures, all information exchanged between two terminals is sent in so-called logical channels. Logical channel is defined in quite an abstract way in the [2] as "Channel used to carry the information streams between two H.323 endpoints. (...) There is no relationship between a logical channel and a physical channel." It is because *ITU-T*

Rec. H.323 determines neither transmission nor network protocol for data exchange. In TCP/IP networks logical channel is a UDP-UDP relationship or a TCP session. Generally, audio and video streams use unreliable protocols (UDP), and data and control streams use reliable protocols (TCP). The control channel is a reliable logical channel used to exchange H.245 control information. There is exactly one control channel between two endpoints. The control channel is bi-directional and is established as the first logical channel in a given session. A regular protocol for opening logical channels does not apply in the case of the control channel. The TCP ports that are used by the control channel are assigned dynamically by both terminals (they are not well-known ports). The called terminal sends a message to the calling one. The message contains, as one of its parameters, the number of a TCP port assigned by the called terminal to the control channel. This number is then used to create a TCP connection by the calling terminal. The details of establishing the channel are described in [5] and [2] and are beyond the scope of this work. All other logical channels in a given session are controlled (opened and closed) using the control channel.

The simple introduction to H.323-related protocols can be found in [8].

1.2.2. Messages and Procedures

Information is transmitted in the control channel in chunks called *messages*. All the messages are defined in an ASN.1 form (refer [3], [4] for details). According to ASN.1, messages are defined by the root of the H.245 logical tree called “Multimedia System Control Message”. Every message is defined by a specific node in this tree. The subnodes of a message’s node are called message’s *parameters* in this work. Some messages may have many complex parameters. If a message is about to be transmitted, it must be transformed from its logical tree into a bit-stream representation (In the H.323 terminal shown in Figure 1-1 this task is accomplished by the C3 module.

There are four types of messages: requests, responses, commands and indications. A *command* is a message that makes the other terminal do something; it does not require any response from the other end. An *Indication* is a message that just informs the other end about the current state of the terminal. It does not require any response or action. A *Request* is a *command* that requires a response. A *response* is just an answer to a request. There are about fifty messages defined in ITU-T Rec. H.245 (refer [1] for details).

Most H.245 messages can be assigned to a procedure. For instance, messages called `OpenLogicalChannel`, `OpenLogicalChannelReject`, `OpenLogicalChannelAcknowledge`, `CloseLogicalChannel` form the set of messages used by the procedure for opening and closing logical channel messages. They are described more thoroughly in Section 1.2.3 and Section 1.3.

The other messages do not belong to any procedure. They may be used separately. The examples include the `FunctionNotUnderstood` indication (sent if message was not understood), and `FunctionNotSupported` indication in case of an unsupported message . Another message of this type is `SendTerminalCapabilitySet` command. Once it is received, the terminal must start capability exchange procedures (share its capabilities with the remote terminal). A session between two terminals is closed as a result of the `EndSession` command. No other messages can be sent after this command; the control channel is closed. There are many more similar commands and indications defined in [1]. The messages that are mandatory in the H.323 terminal are described in Chapter 2.

1.2.3. H.245 Signalling Entities

The request/response messages cannot be used randomly in H.245 protocol. The special sequence of those messages is defined. The algorithms how these messages are used are grouped in special procedures. These procedures are usually quite complex and must take care of timers, protocol errors, non-standard responses, a lack of responses, etc. They cover variety of situations. The H.245 procedures have one more important capability: multiple procedures can work simultaneously. For example, there can be proceeded two or more opening channels negotiations in the same time without interfering each other. To allow that, several “signalling entities” are present in every terminal, which uses the control channel as specified in *ITU-T Rec. H.245*. A signalling

entity defines algorithms of most of the control channel procedures.

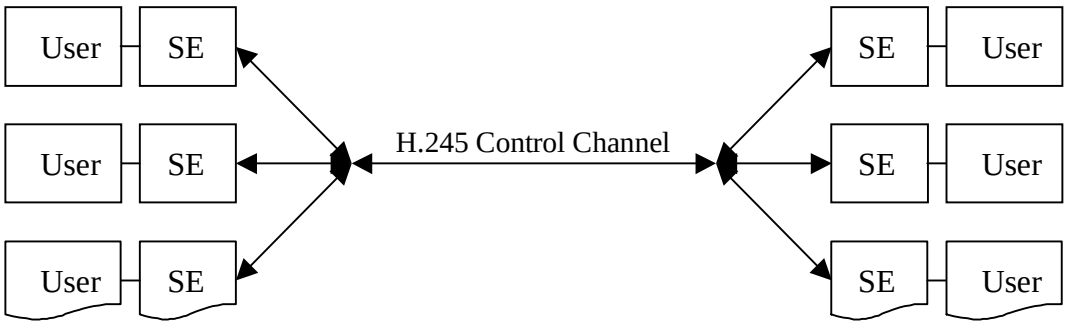

Figure 1-2. Signalling Entities

Figure 1-2 illustrates the H.245 entity system. SE is a signalling entity. User is a module that an entity communicates with. Every entity has a single user assigned to it. And every user has one entity under its control. In the case described here, multiple entities may not be assigned to one user (to allow that entity identifiers must be introduced; but this is an implementation problem).

Every entity and its user have their equivalent at the other side (remote terminal). It is the same type entity. An entity initiating a procedure will be called the local entity. The user of the local entity will be called the local user. The peers will be called the remote entity and the remote user, respectively. The local entity and local user are shown in figures on the left hand side. The *ITU-T Rec. H.245* describes algorithms, data structures, messages and interface for any signalling entity. This behaviour of every entity is called a *procedure* is this work.

Signalling entity is a description of algorithms and variables (like a class in object-oriented programming). Consequently, to make it work, instances of signalling entities are needed. An instance is a separate object, which can be created and destroyed (like an object in OOP). Some signalling entities may have multiple instances; some of them have exactly one. The number of instances and the moments of their creation (and destruction) are explained in Section 1.3.

The terms “signalling entity” and “instance of signalling entity” are used interchangeably below (while it causes no ambiguity).

Any signalling entity must communicate with its peer using messages and with its user using so-called primitives. The user dispatches commissions to its entity in the form of primitives and this makes the entity start required actions (send, receive and analyze messages). The entity answers the user in the form of primitives, too. *ITU-T Rec. H.245* does not define the form of primitives, just their meaning. The primitives contain parameters. These parameters correspond directly to message parameters. For example, while opening a new logical channel, the primitive ESTABLISH.request must be passed by the user to the entity responsible for opening logical channels. This primitive contains a parameter (FORWARD_PARAM) that describes properties of the desired logical channel (especially encoding algorithms). Then a corresponding message (`OpenLogicalChannel`) is prepared by the entity. The message includes a parameter that conveys exactly the same information as FORWARD_PARAM. Then the message is encoded and sent to the peer.

Figure 1-3 shows how signalling entities and their users communicate. RTDSE and LCSE are names of certain entities, they are described later. Primitives are exchanged between entities and users. This communication may be also treated as a direct communication between the users (the local user with the remote user), like in the case of the other stack layers. Messages are sent from an entity to its peer over the control channel.

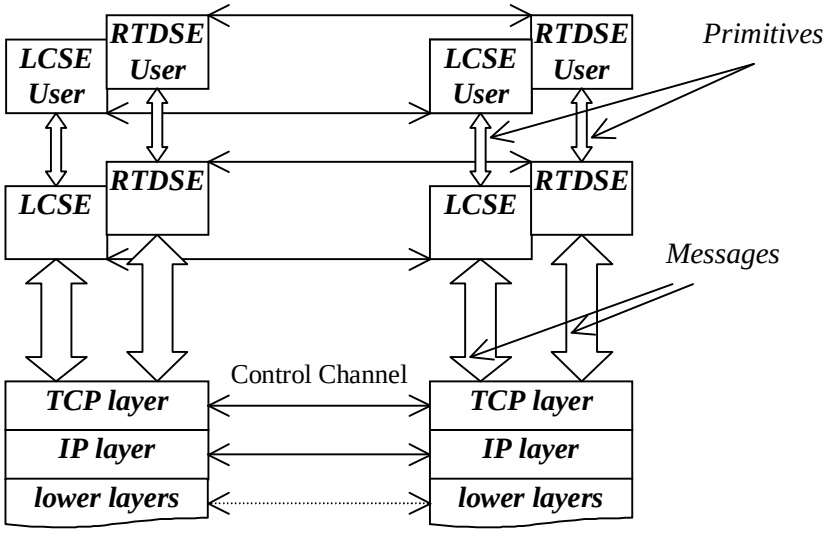

Figure 1-3. H.245 Protocol Stack

All the entities share one control channel but messages are always delivered to the proper entity.

There are four types of primitives: a request, a response, an indication and a confirmation (abbre-

viated in the [1] as *confirm*). They are named using the following rule: "Primitive Task"."Primitive Type", e.g., ESTABLISH.request, ESTABLISH.response, TRANSFER.confirm, etc. A request primitive is passed to a signalling entity to initiate a procedure (e.g., ESTABLISH.request starts an opening logical channel procedure); an indication primitive is passed from the remote entity to the remote user after receiving the first message of the procedure from the local entity. It indicates to the remote user that the procedure was initiated. Usually, once an indication primitive is passed to the remote user, the remote entity starts waiting for the reply from the user (e.g., ESTABLISH.indication primitive asks the remote user whether or not to open a logical channel). If the remote user agrees, it passes a response primitive to the remote entity (e.g., ESTABLISH.response). Confirm primitives are generally passed from the local entity to the local user. They signal that the given procedure has finished successfully (the only exception to this rule is described in Section 1.3.2).

There are two types of signalling entities: outgoing and incoming. An *outgoing entity* is located at the calling side (the side that starts a procedure). An *incoming entity* is located at the called side. In this work a calling entity is named also a local entity, a called entity is named a remote entity. For instance, if the local terminal is going to open a logical channel, the local user responsible for opening the channel, passes the ESTABLISH.request primitive to the outgoing logical channel signalling entity. Next, the message `OpenLogicalChannel` is sent over the control channel to the incoming signalling entity. The outgoing and incoming entities of a procedure operate differently and are described in details in [1].

Some of the signalling entities work as both: outgoing and incoming. In that case the calling and the called side operate the same way.

What is inside a signalling entity? Signalling entities are described as state machines. Receiving a primitive, a message or a timer expiry can change the state of a signalling entity. A state change involves performing some activities. The most common state of almost all the entities is the IDLE state (the state of inactivity). Protocol errors and unsupported functions are handled from outside the entities. Messages not understood are ignored. No user is informed about such a message, and no entity changes its state. If an entity receives a message not appropriate to its state, it must be handled appropriately (e.g., receiving a response message without any previous request message does not change the state of the entity). An entity must be ready for such situations. They cannot make the entity work in an unpredictable way. Most entities are equipped with timers. All timeout errors are passed to the user by the entity in a form of a primitive. After sending a message to the peer, a timer is set. Thus, the entity is prepared for lack of a response; it will be reactivated after some time (usually 5 seconds). Additionally, entities are responsible for supporting counters. If certain activities fail a number of times, they are abandoned. This feature saves an entity from infinite loops.

The following entities have been defined in ITU-T Rec. H.245: Master Slave Determination

Signalling Entity (MSDSE), Capabilities Exchange Signalling Entity (CESE), Uni-directional Logical Channel Signalling Entity (LCSE), Bi-directional Logical Channel Signalling Entity (B-LCSE), Close Logical Channel Signalling Entity (CLCSE), H.223 Multiplex Table Signalling Entity (MTSE), Request Multiplex Table Signalling Entity (RMESE), Mode Request Signalling Entity (MRSE), Round Trip Delay Signalling Entity (RTDSE), Maintenance Loop Signalling Entity (MLSE).

1.3. Description of Entities

This section describes the desired scenarios of procedures performed by entities required by H.323. “Desired” means that they are performed under prosperous circumstances (the remote terminal responds positively to all requests, no timeouts or protocol errors occur, etc.). Diagrams (based on those in [1]) are drawn to illustrate the scenarios. The diagrams use the notation pre-

sented in Figure 1-4.

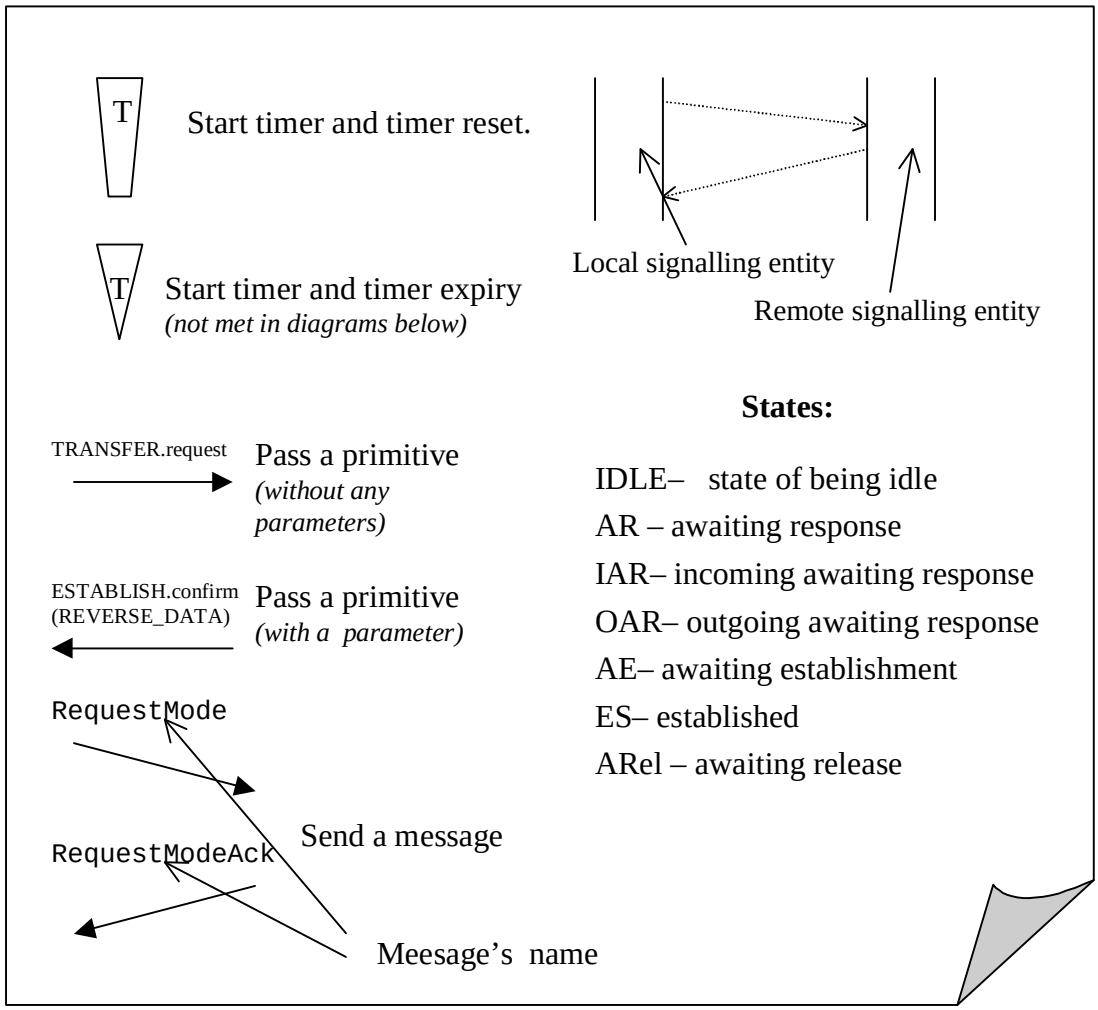

Figure 1-4. Diagram Symbols

Every signalling entity is described with a great emphasis on its interface (communication with the user). All possible primitives are enumerated and the user responsibility is described. Additionally, the parameters of the primitives are thoroughly analyzed.

The entities are introduced in the order they are used in audio-only H.323 terminal. The signalling entities that are not present in H.323 terminal (MTSE, RMESE) and the entities that may be used just for rejecting an every request (MLSE, B-LCSE) are not introduced here.

1.3.1. Capability Exchange Signalling Entity

This procedure is used to exchange terminal's capabilities, especially the protocol version and supported codecs.

To understand the essence of this entity the way of storing terminal capabilities must be introduced. There are three types of capabilities: transmit, receive and "transmit and receive" capabilities. Transmit and receive capabilities are separated - they do not interfere. If the "transmit and receive" capabilities are present, that means that the terminal has some dependencies between transmit and receive logical channels. For instance, it can either transmit or receive video streams but never do both at the same time.

Transmit or receive capabilities may be absent in primitive parameters (and consequently in message parameters). Omitting the receive parameters implies that the terminal cannot receive any multimedia signals. If the transmit capabilities are omitted, the terminal may still be a transmitter but does not offer any preferred modes (e.g., for the MRSE purposes, described later).

The data structures described below are identical for transmit, receive and "transmit and receive" capabilities. In this case it is supposed that these are receive capabilities.

All the audio/video/data/encryption capabilities are enumerated in a table called a capability table. Every item in this table has two fields - a capability number and a capability name: (NUMBER, CAPABILITY). This table may be treated as a space of basic capabilities. For example, {(1, G.711), (2, G.722), (3, H.261)} would be a valid capability table. A subset of this space called an alternative capability set. Its meaning is that every capability from the subset may be used alternatively (exactly one from the set may be used at any given time). For instance, for the previously defined capability table the following alternative capability set may be defined among others: {1, 2, 3} (any of the codecs from the capability table can be used) and {1, 2} (just G.711 and G.722 can be used). A family of alterative capability sets is called a simultaneous capability table. Let T={{1,3}, {2,3}}. This simultaneous capability table (T) consists of two sets A={1, 3} and B={2, 3}. It describes what codec combinations can be used. To find out the combinations, the Cartesian product of the sets in T must be calculated. In this case AxB={{1,2}, {1,3}, {3,2}, {3,3}}. A set of simultaneous capability tables is called simultaneous descriptors. And this is the parameter describing capabilities of the terminal.

Notice that *the CESE procedures must precede all other procedures in the control channel.*

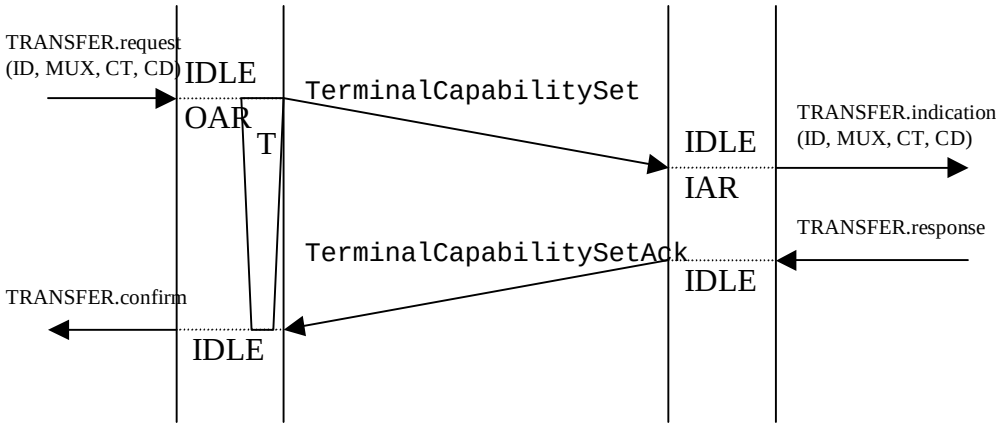

Figure 1-5. Capability Exchange (CESE)

Figure 1-5 shows an exchange of messages and primitives for the CESE procedure. They are started when the local user passes the TRANSFER.request primitive to the local entity. This request includes four parameters: identifier of the used H.245 protocol used (ID), a multiplex capability table (MC) (not used in H.323 terminals), a capability table (CT) and capability descriptors (CD). They may contain transmit and receive capabilities for any media types. Once the primitive is passed to the local entity, the `TerminalCapabilitySet` message is sent to the remote entity. This message contains all parameters passed in the TRANSFER.request primitive. The remote entity receives the message and passes the TRANSFER.indication primitive to the remote user. The primitive contains all the parameters acquired from the message. The remote user responds using the TRANSFER.response primitive. Next, the remote entity sends the `TerminalCapabilitySetAck` message to the local entity. When the local entity gets this message, the TRANSFER.confirm primitive is passed to the local user.

Instances: There are two pairs of instances for every call. During a normal session (two terminals are involved) a terminal has only one instance of incoming CESE and one instance of outgoing CESE.

Other primitives: REJECT.request(CAUSE) primitive can be the answer of the remote user to the TRANSFER.indication primitive. CAUSE must be filled with the cause of rejection. The local user gets the REJECT.indication(CAUSE) primitive instead of the TRANSFER.confirm one.

Responsibility of the user: The local user must properly fill the parameters and then pass them to the entity. The remote user should store the received capabilities.

1.3.2. Master Slave Determination Signalling Entity

This procedure is used to designate one terminal as master, and the other as slave. The assignment is used by some other entities (e.g., LCSE, B-LCSE).

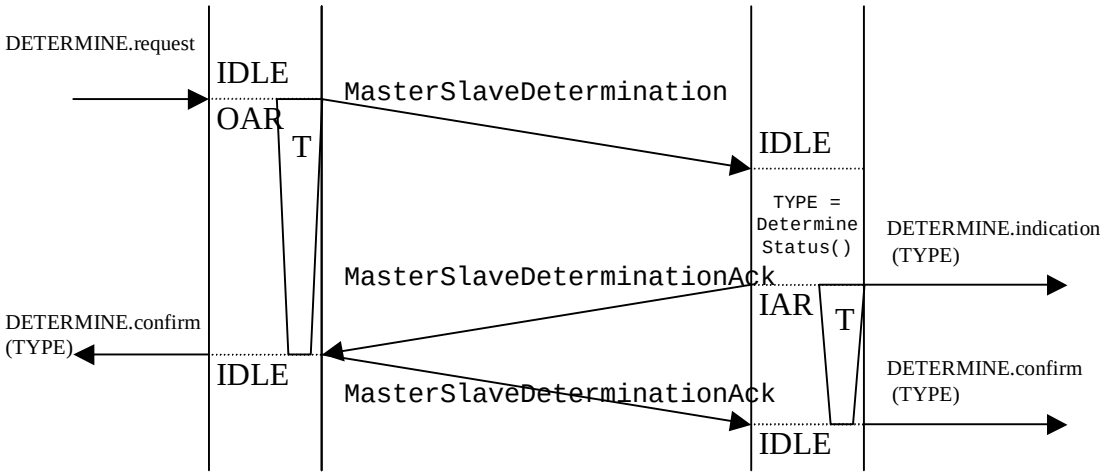

Figure 1-6. Master Slave Determination (MSDSE)

First, as illustrated in Figure 1-6 the local user passes the DETERMINE.request primitive to the local entity (no parameters). Then the entity sends the `MasterSlaveDetermination` message to the remote entity. This message contains two important parameters: the terminal type (TT) and the status determination number (SDNUM). TT is a number assigned to every terminal. This value is specified in the recommendations that refer to ITU-T Rec. H.245. Generally speaking, if an entity supports more features, the TT number is greater (e.g., a terminal with no MC has the TT number equal 60, a gatekeeper with no MP has TT number equal 120, while an MCU with data, audio and video has TT number equal 190). SDNUM is a 24-bit unsigned integer random number generated by the local entity. The remote entity receives the message and runs the `determine_status` function. The function first compares TT from the message with the local TT. The terminal with the bigger TT is picked as the master terminal. If both TT's are equal, two SDNUMs (received and local) are compared. The `determine_startus` function returns one of the following values: "master", "slave", and occasionally "undetermined" (when an unusual coincidence happens). The returned value is passed to the remote user in the DETERMINE.indication primitive and stored in the entity's local variable `state`. This primitive informs the user about the results of the master slave determination. Then, the `MasterSlaveDeterminationAck` message with the inverted `state` variable as its parameter is sent to the local terminal. A local variable `state` at the local entity is set, and the DETERMINE.confirm primitive is passed to the local

user. Then the `MasterSlaveDeterminationAck` message is sent to the remote entity with the `state` value inverted again. The peer receives the message, and if the `state` local variable equals the message parameter `state`, the primitive DETERMINE.confirm is passed to the remote user. Now the user is informed that both the local and the peer side have resolved the issue successfully.

Instances: There is just one instance of this entity in every terminal. This instance shares outgoing and incoming features (calls and can be called).

Other primitives: the REJECT.indication primitive may be passed by the entity to the user if it acquired the `MasterSlaveDeterminatonRelease` message or the `MasterSlaveDeterminatonReject` message. These are caused by the simultaneous initiation of MSDSE procedures. The users can also get the ERROR.indication caused by undetermined result of the `determine_startus` function or by the timer expiry.

Responsibility of the user: The local user as well as the remote user must store the current status somewhere (master, slave, undetermined).

1.3.3. Uni-directional Logical Channel Signalling Entity

This procedure is used for opening and closing uni-directional (the commonest) logical chan-

nels.

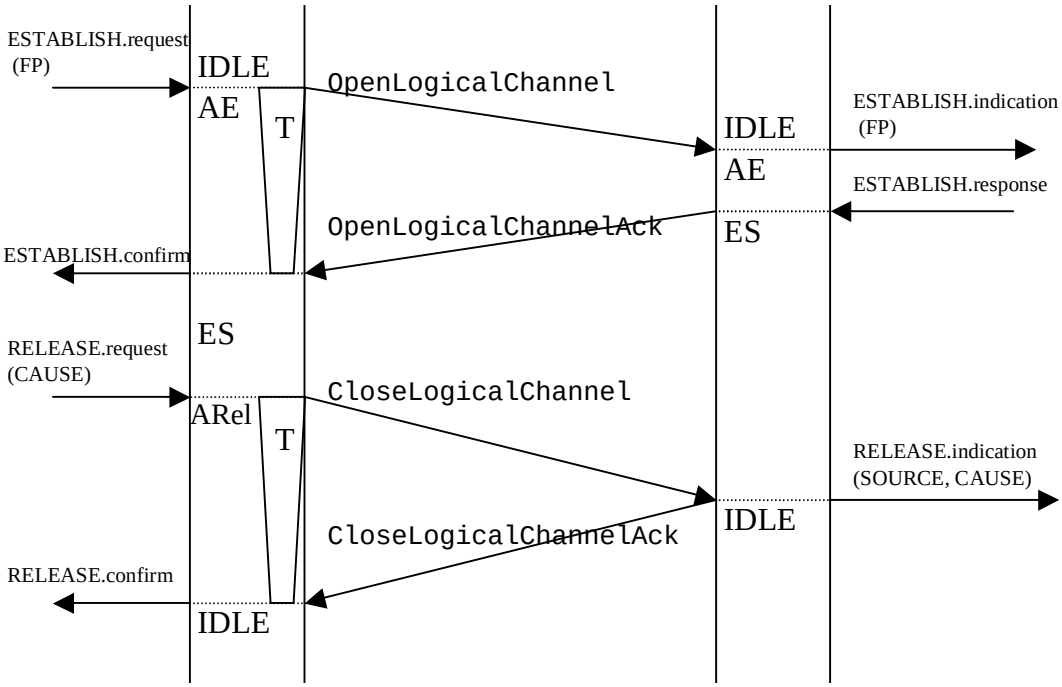

Figure 1-7. Logical Channel Signalling Entity (LCSE)

Figure 1-7 illustrates how a logical channel is opened and closed. First, the local user passes the ESTABLISH.request(FP) primitive to the local entity. The primitive contains forward parameters (FP). FP is a very complex set of logical channel parameters. It contains the number of the channel to be opened, assigned arbitrarily by the transmitter. It contains a UDP port number for the media-carrying RTP logical channel that is being opened and a UDP port number for the reverse RTCP. Finally, FP contains the data type (e.g., audio), a codec used and, in case of some codecs, some additional parameters. These parameters correspond directly to parameters of the `OpenLogicalChannel` message, which is sent to the peer. The remote entity gets the message and passes the ESTABLISH.indication primitive based on the parameters from the message to the remote user. Then it the remote entity waits for an answer. If the remote user agrees to open such a logical channel, it passes the ESTABLISH.response primitive to the entity and that makes the entity send the `OpenLogicalChannelAck` message to the local entity. After receiving this message, the local entity passes the ESTABLISH.confirm primitive to the local user. The logical channel is now in ESTABLISHED state, and the transmission may begin.

To close the logical channel, the local user passes the RELEASE.request primitive to the local entity (no parameters). The transmission of data must be terminated before passing this primi-

tive. The `CloseLogicalChannel` message is sent to the peer. The remote entity gets the message and immediately responses with the `CloseLogicalChannelAck` message, sending the RELEASE.indication primitive to the remote user at the same time. The local entity passes the RELEASE.confirm primitive to the local user. The logical channel is now closed.

Instances: There is the pair of instances for every opening channel procedure. The local entity uses the outgoing algorithms, and the remote entity - incoming algorithms. The number of instances varies from 0 to N (N is a number of logical channels). New instances are created with the ESTABLISH.request and should be killed after the associated logical channel is closed.

Other primitives: the RELEASE.request(CAUSE) primitive may be passed by the remote user to tell that it does not agree to open the logical channel. The CAUSE parameter must be set. The local user then receives the RELEASE.indication primitive. Another primitive is the ERROR.indication(ERRCODE) primitive meaning that the timer expiry or a protocol error has occurred.

Responsibility of the user: In case of this entity the local user must be aware of many factors; also it must perform some actions before starting the LCSE procedure. First, the user must select a proper codec. Second, the user must check in the capability table (received from the remote peer as a result of the CESE procedure) whether the remote terminal supports the chosen codec. Third, the user must check if this codec can work simultaneously with the already opened logical channels (looking up the simultaneous capability tables). Next, the user must check whether bandwidth (allocated by a gatekeeper and received from RAS) is not exceeding due to using the new logical channel. Then a UDP socket (if opening a video or audio channel) for further media exchange (RTP) and a UDP socket (for the RTCP protocol) must be opened. RTCP is not present in all types of logical channels; sometimes this parameter may be suppressed. Next, proper forward parameters (FP) must be formed. For some sets of codecs, special parameters must be added (e.g., a dynamic payload type for RTP purposes or a silence suppression flag).

Once the ESTABLISH.indication primitive is received by the remote user it checks whether it can open an additional logical channel with the given parameters. If positive, it opens two UPD sockets: RTP and RTCP one. Next, the remote user must be ready to receive the media stream (must activate a media engine new threads or processes). Then the `OpenLogicalChannelAcknowledge` message may be sent to the local entity.

1.3.4. Close Logical Channel Signalling Entity

The following procedure is used to request the closing of logical channels by the receiver. Normally, the transmitter opens and closes its logical channels. But sometimes the receiver cannot process a media stream or no longer wants to receive it. In these cases the CLCSE procedure must

be performed.

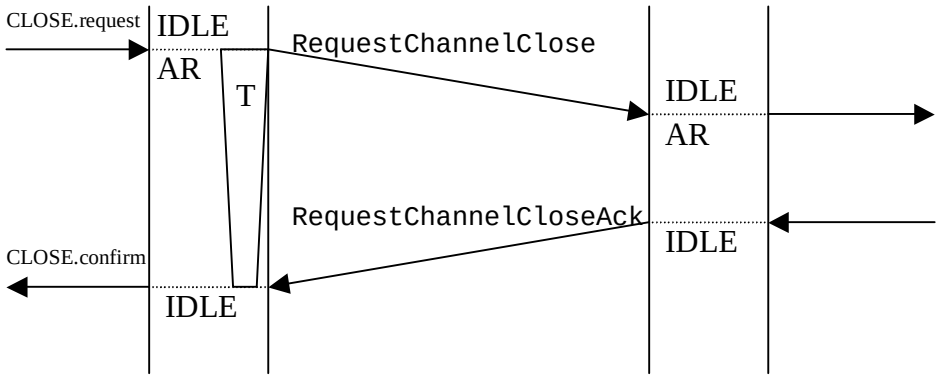

Figure 1-8. Close Logical Channel Signalling Entity (CLCSE)

Figure 1-8 shows the CLCSE procedure. In this case the local entity is a receiver, the remote entity is a transmitter. First, as illustrated in the figure, the local user passes the CLOSE.request primitive to the local entity. Then, the entity sends the `CloseLogicalChannel` message to the peer (the message includes the number of the logical channel). The message is received by the peer (the remote entity) and the remote entity passes the CLOSE.indication primitive to the remote user. If the remote user agrees to the suggestion in the primitive, it passes the CLOSE.response primitive to the remote entity. The remote user is now obliged to start closing this logical channel (passing the RELEASE.request primitive of the LCSE procedure to the proper LCSE entity). Then the remote (incoming) entity sends the `CloseLogicalChannelAck` message to the peer. The local entity gets the message and passes the CLOSE.confirm primitive to the local user.

Instances: The Recommendation specifies that there is exactly one outgoing instance for every receiving logical channel and one incoming instance for every transmitting logical channel. Consequently, the number of the instances is equal to the number of all the opened logical channels. They are created once LCSE procedures are started, and destroyed once the associated logical channel is closed.

Other primitives: the REJECT.request(CAUSE) primitive may be passed by the remote user to the remote entity if it does not agree to close the logical channel. CAUSE parameter must be set. The local terminal receives the REJECT.indication(SOURCE, CAUSE) primitive in this case (the value of SOURCE is “USER”). The REJECT.indication primitive is passed to the local user after the timer expiry inside the local entity (SOURCE has the value “PROTOCOL” in this case).

Responsibility of the user: The local user has no special responsibilities. The remote user must

start the closing part of the LCSE procedure after successful completion of the CLCSE procedure.

1.3.5. Mode Request Signalling Entity

These functions are used to initialize the opening of logical channel by the receiver. The receiver does not open the logical channel. It just sends a polite request to the transmitter to open a desired logical channel. Two entities are present in the time of running the procedures: incoming and outgoing (starts them). After a successful finishing of these procedures, the incoming peer must start the opening logical channel procedures. Notice that if there were no transmitting capabilities sent by the remote terminal while CESE procedures it means that the RMSE procedures must not be used.

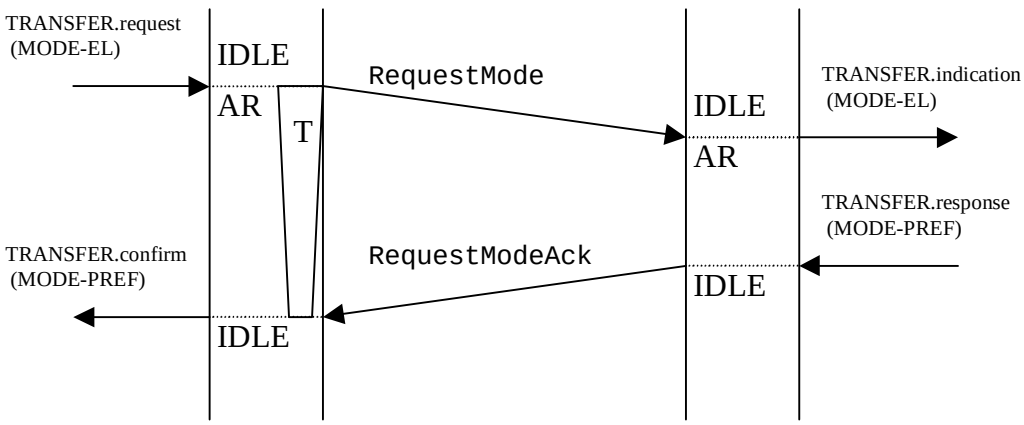

Figure 1-9. Mode Request Signalling Entity (MRSE)

First, as shows Figure 1-9, the TRANSFER.request primitive is passed to the local entity. The primitive carries the parameter called mode element (MODE-EL). This parameter contains the enumerated modes from the most preferred to the least preferred mode. The modes must conform to the capability table (received while CESE procedures). Then the entity sends `RequestMode` to the remote terminal. This makes the remote entity pass the TRANSFER.indication primitive to the remote user. If the remote user responses positively, it passes the TRANSFER.response primitive containing information whether the most proffered mode was chosen. Then, the remote entity sends `preferred` to the local entity. The local entity acquires the message and passes the TRANSFER.confirm primitive to the user.

Instances: There is the pair of instances for every call. The local entity uses the outgoing algorithms, and the remote entity - incoming algorithms. During an ordinary, two terminal session

there is one instance of outgoing and one instance of incoming CESE.

Other primitives: the REJECT.request(CAUSE) primitive may be passed by the remote user to the remote entity if it does not agrees to open new logical channel. CAUSE parameter must be filled. The local terminal receives the REJECT.indication(SOURCE, CAUSE) primitive in this case (the value of SOURCE is “USER”). The REJECT.indication primitive is passed to the local user after the timer expiry error inside the local entity (SOURCE has the value “PROTOCOL” in this case).

Responsibility of the user: The local user must refer to the items in the capability table (CT) and in the capability descriptors (CD) set before starting these procedures. The remote user must store the chosen mode after its positive response to the entity. It must start LCSE procedures immediately after completion of the MRSE procedures using the stored mode.

1.3.6. Round Trip Delay Signalling Entity

Round Trip Delay is used to check the delay in the control channel. The delay is measured by the local entity as the time between sending a message and receiving the response for this message.

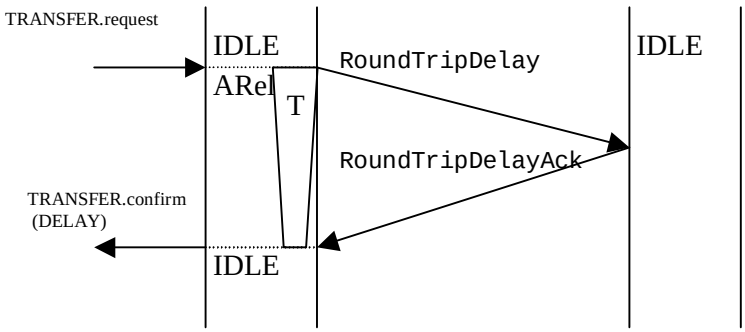

Figure 1-10. Round Trip Delay Signalling Entity (RTDSE)

The procedures (refer Figure 1-10) are started by the TRANSFER.request primitive passed to the local entity. Then, the `RoundTripDelay` message is sent to the remote terminal and the timer is started. After getting this message, the remote entity answers immediately with the `RoundTripDelayAck` message. This message, when acquired by the local entity, stops the timer and makes the entity pass the TRANSFER.confirm primitive with the properly calculated delay as a parameter (DELAY).

Instances: Only one instance of RTDSE must be present all the time. It shares the incoming and outgoing capabilities.

Other primitives: EXPIRY.indication is passed to the local user when no response from the remote terminal was acquired in the given time.

Responsibility of the user: The local user should store the DELAY value after the positive completion of the procedures. The remote user is not informed about incoming RTDSE procedures.

Chapter 2. H.245 in H.323.

This paragraph describes all the H.245 features, which are mandatory in an H.323-compatible terminal. First, H.245 control channel is established immediately after successful procedures of H.225.0 Call Signalling. The essential information is exchanged during these procedures (e.g. the TCP port for the control channel). Second, every H.323 terminal must be able to parse all the H.245 messages (even if it does not support them). It is important to understand and response properly for such messages, more adequate than just `Function Not Understood` indication. Finally, the control channel (with an identifier 0 assigned to it) must be permanently open and usable until the entire session is finished.

As mentioned earlier, an H.323 terminal uses just a subset of all H.245 protocol features. *ITU-T Rec. H.323* gives lists of signalling entities, which are present in terminals as well as messages, which must be understood by such endpoints. In this set of signalling entities there are entities, which have some messages forbidden or optional (i.e. endpoint does not have to support them). The required entities are: MSDSE, CESE, LCSE (some messages are optional), B-LCSE, CLCSE (some messages are optional), RMSE (some messages are optional), RTDSE (some messages are optional) and MLSE (almost all messages are optional). For details, refer [2], p. 19, p. 82 .

Bi-directional LCSE (B-LCSE) is not used in speech-only terminals to establish logical channels (because bi-directional channels are not present in such terminals). This entity uses the same set of messages as LCSE (with additional parameters). If a speech-only terminal acquires a request to open a new bi-directional channel, it should refuse.

Maintanance Loop Signalling Entity (MLSE) is required for every H.323 terminal but most messages and their parameters are optional or forbidden. System loops and logical channel loops are forbidden. Just media loops only are permitted though optional (consequently video or audio streams may be looped and than come back to the transmitter). The mandatory messages are: `MaintenanceLoopOffCommand` (turning off all the loops) at the receiving side and `MaintenanceLoopReject` at the transmitting side.

Every terminal must support the set of commands and indications (not assigned to any signalling entity). The commands that must be implemented at the transmitting and receiving sides are: `SendTerminalCapabilitySet`, used to make the peer start CESE procedures, and `EndSession` used to end the session between two terminals. There are several other commands that must be implemented at one of the sides are not described here (refer [2] p. 86). The mandatory indications are `FunctionNotUnderstood` and `FunctionNotSupported` and `UserInput` command. `UserInput` is a command that allows sending to the peer some characters as if they were typed on a keypad. The minimal set of character is 0-9, * and #.

A logical channel, opened using LCSE procedures, has an audio/video/data codec assigned to

it. Every corresponding RTP stream has a payload type assigned to it (according to [6] and [7]). There must be a correlation between this payload type and the H.245 codec type.

Every H.245 protocol revision has a version number. [1] describes the version number 2 of the protocol. This protocol *must* be compatible with all preceding versions (in this case with version number 1).

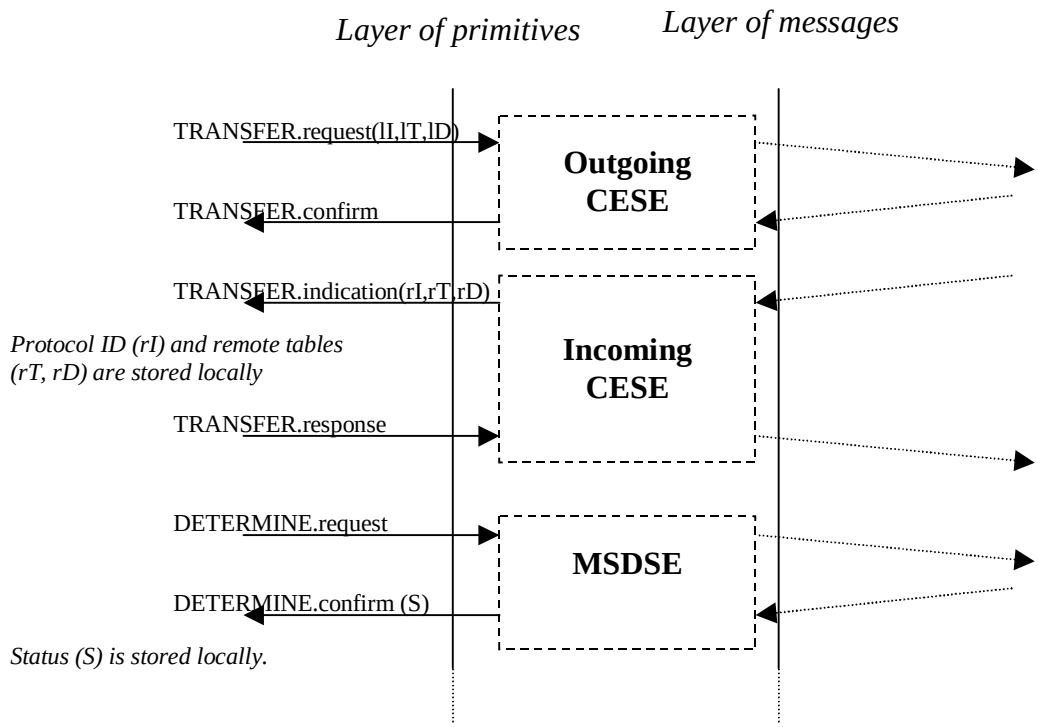

Figure 2-1. H.245 Initial Sequence

Figure 2-1 shows the beginning of the H.245 session. The first messages sent into control channel are CESE messages. These give the remote terminal information about the local terminal capabilities. The H.245 protocol version is transmitted, and then all supported audio/video/ data capabilities are sent (in capability tables and capability descriptors). At the every moment in the particular session, terminals may sent the command `Send Terminal Capability Set` which makes remote terminal start CESE procedures and send all the interesting parameters. The figure shows that outgoing CESE procedures are run once per session. Incoming procedures (initiated by the remote terminal) are run also just once. Normally, there is no need to perform these procedures several times (the terminals have constant capabilities). But if the capabilities change in

time (for example the user turns the video capabilities off to make up) the CESE procedures may be performed also in the other terminal states. So the terminal must be ready for such a situation.

Next procedures are MSDSE ones. It is a must to assign one terminal as a master terminal and the other as a slave terminal before the opening of logical channel if performed. Some conflicts may be met while establishing a logical channel. In such cases the master terminals gains some profits (the slave terminal must give up). The MSDSE procedures, if successful, should be performed just once, after the CESE procedures. Nevertheless, MSDSE entity is active all the time during the session. Every terminal must be able to work in master as well as slave modes.

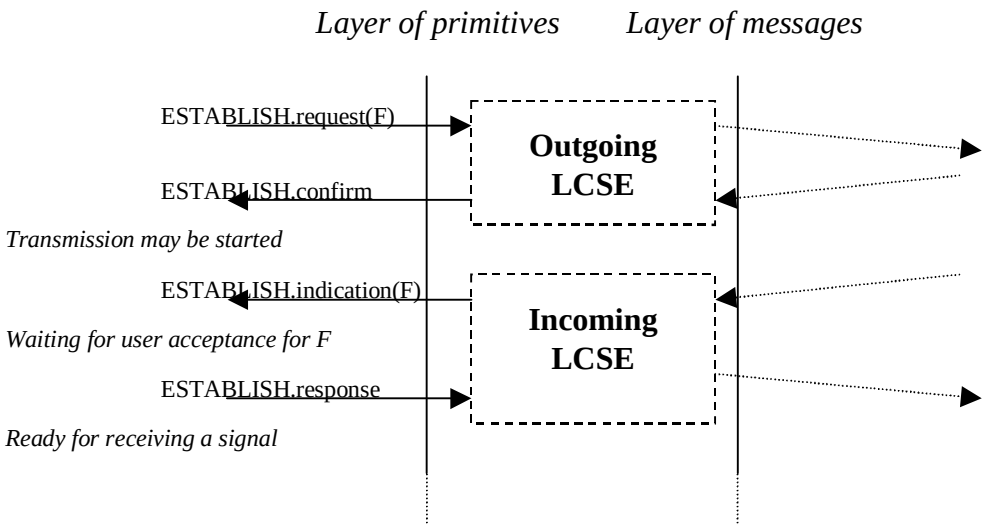

Figure 2-2. Opening logical channels

In the moment the MSDSE results are known for both of the terminals (the DETERMINE.confirm primitive is received by the users) LCSE procedures may be started. The number of logical channels depends on the particular situation and is not defined by the Recommendation. Perhaps, it may be needed to start RMSE to open all desired logical channels. In the considered situation (illustrated in Figure 2-2 two logical channels are opened (supposedly they are audio channels): first is opened by the local terminal and the other by the remote terminal. No additional functions are needed in this case.

Once the transmitting logical channel is opened (it is in the ESTABLISHED state) the transmission may be started. In the figured case no additional primitives are sent, but the H.245 subsystem is still alive, and the regular procedures may be performed (e.g. CESE, RTDSE, LCSE, CLCSE, ..). The Recommendation requires from the terminal to handle all the messages in all states.

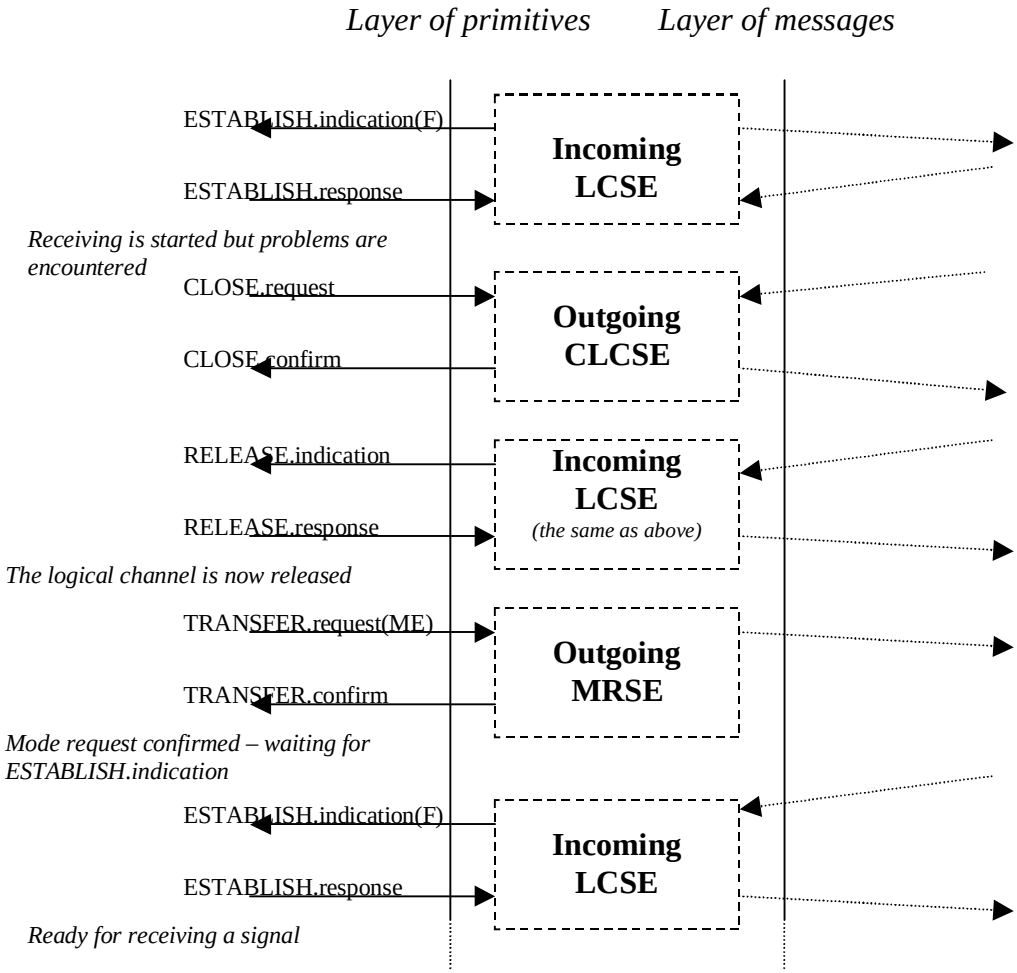

Figure 2-3. CLCSE and MRSE procedures

Occasionally, an H.323 terminal can use additional mechanisms to modify logical channels. This example illustrates the situation where a terminal decides to change the logical parameters, which are not adequate for the terminal. As shown in Figure 2-3 receiving logical channel is opened on the remote terminal's own initiative. The local terminal begins receiving the audio signal but it finds out that in cannot process the signal properly (e.g. too complex decoding rules for the local CPU). Consequently, the local terminal starts CLCSE procedures to request the closing the problematic logical channel. The remote terminal agrees to this request (CLOSE.confirm) and then starts releasing procedures (RELEASE.indication). Then, when the logical channel is eventually released, the local terminal decides to open a new logical channel (with all the parameters carefully selected) and start MRSE procedures (sending TRANSFER.request(ME) to the outgoing MRSE).

The remote terminal agrees again with the local terminal's suggestion and starts LCSE procedures. This is the way to "change" logical channel's parameters.

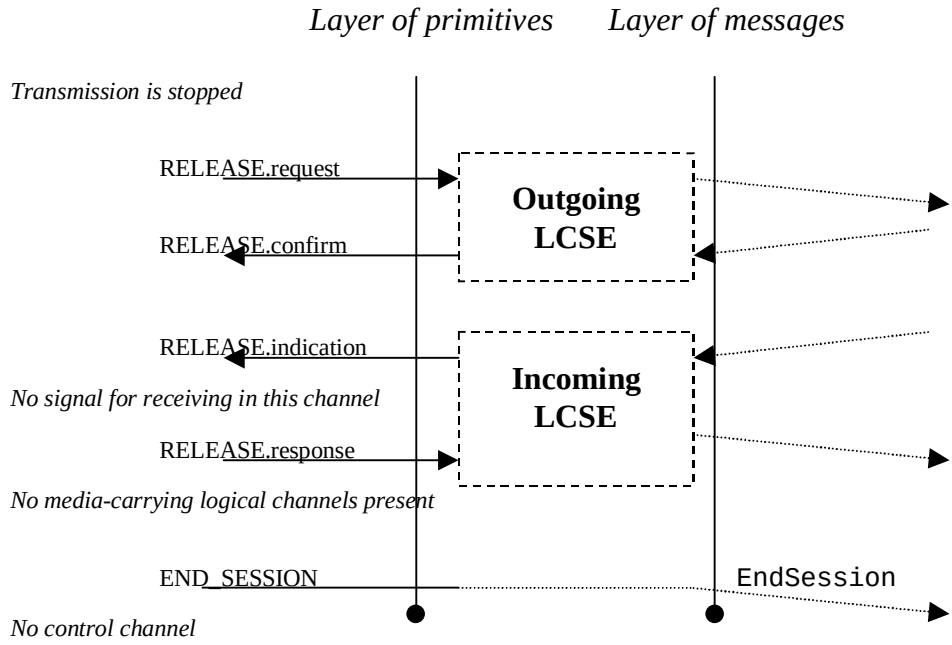

Figure 2-4. Closing the connection

If the terminal is going to terminate the connection is should stop all the media streams, and close all the transmitting logical channels. Then it sends `EndSession` messages to the terminal (represented here by END_SESSION primitive). In this moment the connection is closed, and control channel exists no longer. The Recommendation does not require from the terminal to close all the logical channels before sending `EndSession`, but just suggests it. Consequently, the "ugly" closing of connection procedure would include just sending of `EndSession` message.

Chapter 3. H.245 Subsystem Implementation

3.1. Architecture

In this H.323 terminal project all signalling entities are controlled by the same user (called *Supervisory Thread*). From such a point of view, H.245 Subsystem is treated as a monolith. There is no need to distinguish separate signalling entities' instances.

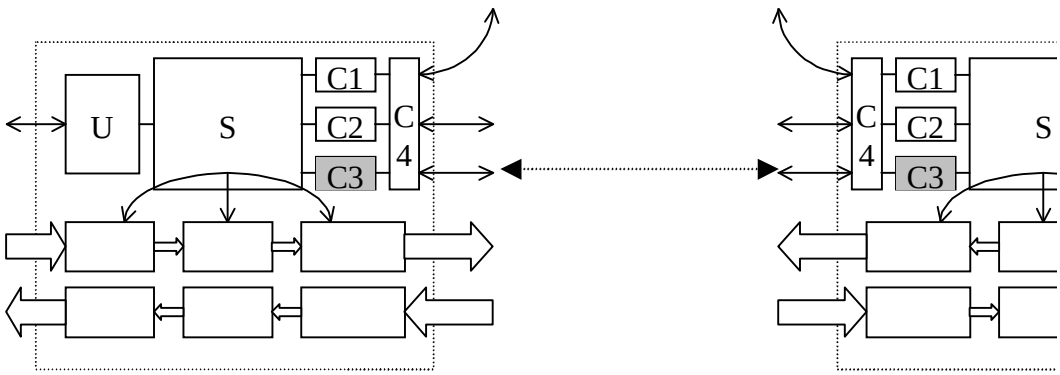

Figure 3-1. H.245 Subsystem

As shows Figure 3-1, the Supervisory Thread (*S* in the figure) passes primitives to the *H.245 Subsystem* (*C3* in the figure) and the Subsystem responses (in case of outgoing procedure) or the H.245 Subsystem passes primitives to the supervisory thread (incoming procedures). The Supervisory Thread must be ready for signals from the *User Control Module* (*U* in the figure), which are ordered by the human user. At the peer side there is a similar functionality.

A primitive passed by the Supervisory Thread is delivered to the proper entity, which is previously created if needed. Unlike in Figure 1-2, signalling entities are not connected directly to the TCP and to the user. Creation and choosing the proper entity is performed by two threads called "Managers".

Note that, *the responsibility of the H.245 user (as described in the previous chapter) is in this implementation divided among the managers and the supervisory thread. The primitives described here are slightly modified comparing to H.245 primitives : to distinguish every entity instance,*

some additional parameters were introduced. Commands and indications have the same form as the primitives passed to the entities.

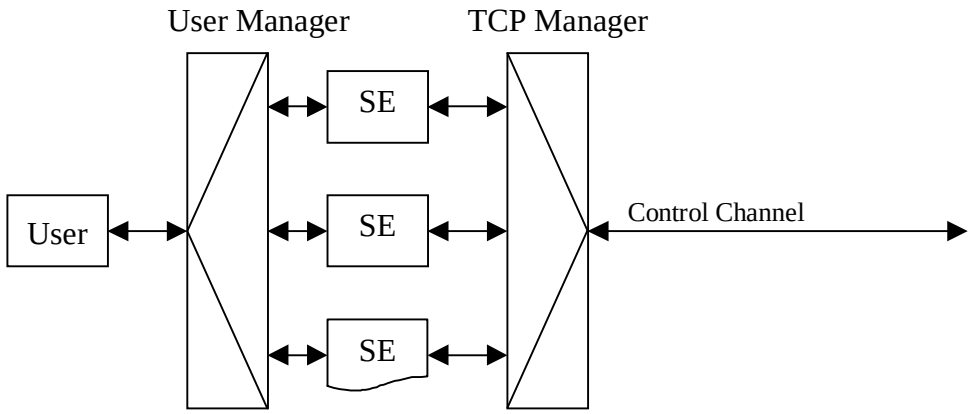

Figure 3-2. H.245 Managers

The first manager, the “User Manager”, is responsible for communication with the user; the second one, “TCP Manager”, communicates with other terminals over a TCP socket (Figure 3-2). Primitive manager is a looped thread, being blocked on queue of primitives almost all the time. When the Supervisory Thread passes a primitive, the User Manager gets it and starts analyzing. It checks whether it is a system control primitive, making manager (and all H.245 subsystem) terminate. If it is so, all H.245 threads are killed and H.245 is stopped. If not, the User Manager looks whether the primitive can be cast to simple message (command or indication). In this case, a message is prepared and sent to remote terminal. Otherwise, a primitive must be passed to a signalling entity. First, the type of entity is determined. Next, decides whether create new instance of the entity; if so, new entity instance is created. Then, primitive is sent directly to this entity. Entity instances are implemented as separate threads, created by managers. If the entity instance is no longer needed, it destroys itself. Entities sent primitives and messages directly to the user and to the remote terminal, respectively. The (C-like) pseudocode of the entity is presented in Figure 3-3.

```
while( !stop_condition() )
```

```
{
  p = get_a_primitive_from_the_queue(); /* blocking function */

  if ( is_a_special_primitive(p) ) {

       take_appropriate_action(p); /* e.g. Terminate Manager */

  } else if ( is_command_or_indication(p) ) {

        m = prepare_an_appropriate_message(p);
        send_the_message_to_the_peer(m);

  } else if ( is_an_h245_primitive(p) ) {

        e = determine_entity_type(p);
        if (new_instance_must_be_created(p) ) {

                i = create_new_instance(e);

 } else { /* The instance exists */

                i = get_instance(p);
 }

        pass_primitive_to_the_instance(i, p);
  }

} /* while */
```

Figure 3-3. User Manager Pseudocode

The second manager, the TCP Manager, is also a looped thread, but it blocks on the Control Channel's TCP socket, waiting for messages from the remote terminal. First, it looks whether it is a command or indication message. If positive, the message is directly cast into a command primitive or indication primitive. If not, the message is passed to a proper entity instance (the instance is created if needed). The pseudocode of this manager is presented in Figure 3-4.

```
while( !stop_condition() )
{
  m = get_a_message_from_an_assigned_socket(); /* blocking function */
```

```
  if ( is_command_or_indication(m) ) {

        p = prepare_an_appropriate_primitive(m);
        pass_primitive_to_the_suprevisory_thread(p);

  } else if ( is_a_well-known_request_or_response(m) ){

        e = determine_entity_type(m);

        if ( new_instance_must_be_created(m) ) {

                i = create_instance(m);

 } else { /* The instance exists */

                i = get_instance(m);
 }

        pass_the_message_to_instance(i, m);

   } else if ( is_an_unknown_message(m) ){

        m1 = prepare_message_parameters("FunctionNotUnderstood", m);
        send_the_message_to_the_peer(m1);
  }
} /* while */
```

Figure 3-4. TCP Manager Pseudocode

After passing a primitive to the entity, it starts analyzing the primitive and its parameters and, depending on the internal state of the entity, appropriate action is taken. This first s.pdf of the action are usually:

- A fill-up of message's parameters,
- A ASN.1 message's encoding,
- Starting of a timer,
- Sending a message to the peer.

Next, entity waits for remote terminal's response. If there is no response, then an error primitive is sent to the user. In many cases, the response from the remote terminal makes the entity send a

primitive to the user, and start waiting for a user's answer. Some entities' instances are destroyed after finishing their job.

The exact internals of designed H.245 subsystem are shown at Figure 3-5.

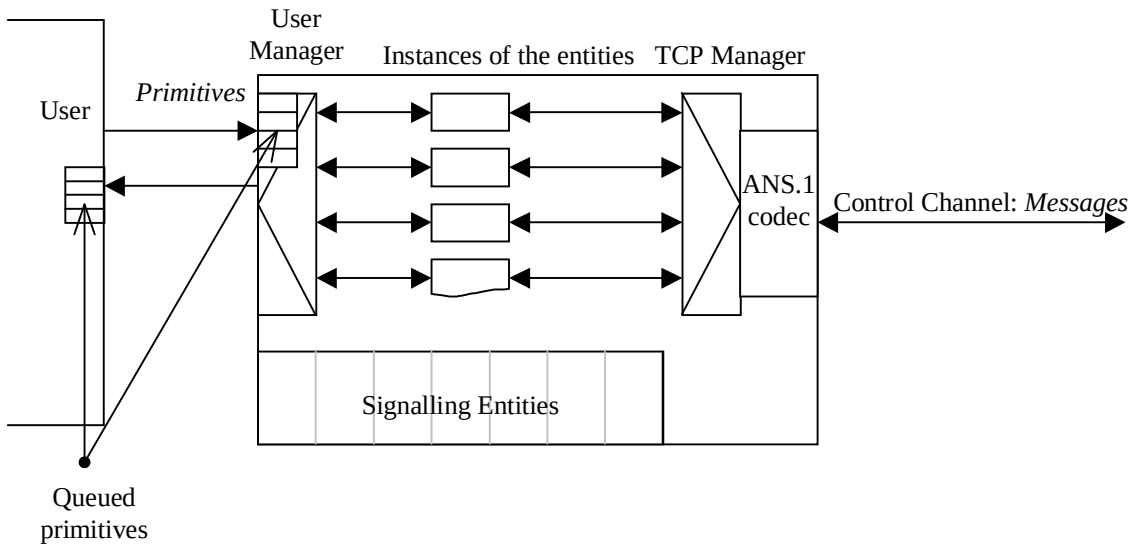

Figure 3-5. H.245 Subsystem Internals

At the left side of Figure 3-5 there is the user, communicating with the subsystem. Primitives are queued in special queues. The most often, the managers have to parse the message, to create the entity, and to prepare the proper primitives for the entity. It takes some time and if the primitives were not queued, some of them would be lost. Functions sending and receiving primitives are just functions which put and get nodes from the queues. At the bottom of the figure there are signalling entities starting procedures - if one of the managers wants to create a new instance of the procedure, it just calls the proper function. Outgoing entities are creating by the "User Manager", incoming ones by the "TCP Manager". Once an entity instance is created, it can send and receive messages as well as primitives. Some of the entities must be created while creation of the entire subsystem (MSDSE and RTDSE) .

3.2. Implementing Entities

Entities are organized according to [1]. The recommendation introduces the SDL diagrams to describe the algorithms of how the signalling entities work. These diagrams in this project are transformed directly into C-code. There is no need to introduce the entire bodies of entities' procedures. Instead, the rules of how SDL diagrams were converted into the code. All the rules and the code were prepared by the author. Of course, the transformation could be organized in a different way. The used structures are presented in The API, used by the entities is presented in . Four special functions have been prepared to be used by the procedures. These functions are called from the entities' functions, so they are internal ones. The user does not need to be conscious of their capabilities. They are introduced here to show how the conversation is preceeded.

The `SendH245Message` function is used to send a message to the peer (over a TCP socket). *`mn`* is a name of the message (e.g. `OpenLogicalChannel`), and *`parameters`* are parameters of the message. This function prepares the message (ASN.1 tree), then encodes it and sends to the peer.

The `SendPrimitive` function passes a primitive to the user. *`mt`* is a name of a primitive (e.g. `H245_OPEN_LC_CONF`), and *`message_params`* is a structure of the parameters for the given message.

The `GetMessageOrPrimitive` function gets a message or a primitive. When a message comes from the peer, it is stored in the entity message queue by the "TCP Manager". Every entity has a pointer to this queue as well as it has a pointer to the queue of primitives from the user. Say that two variables `q1` and `q2` are the pointers to queue from the "TCP Manager" and from the "User Manager" respectively, and `m`, `n`, and `o` are variables of the type struct message. `GetMessageOr-Primitive` may be called as follows:

```
m = GetMessageOrPrimitive(q1, q2);
n = GetMessageOrPrimitive(q1, NULL);
o = GetMessageOrPrimitive(NULL, q2);
```

The first line shows the typical call of the routine (two non-zero parameters). First, the function checks whether there are any messages in `q1`. If positive, function returns with the appropriate message. Otherwise it checks `q2`. If positive, it returns the message. Otherwise the function blocks infinitely until a message appears in any queue. The second and the third call from the example show the usage of the function with one parameter only. In this case the functions checks just one, non-zero queue.

The `GetMessageOrPrimitiveWithTimeout` function is the version of `GetMessageOrPrimitive` with a timeout. Unless it can get a primitive from any queue, it blocks for *`secs`* seconds. After this time it returns a special, "timeout" primitive.

Now the transformation of SDL diagrams may be introduced. Figure 3-6 shows how every block is interpreted.

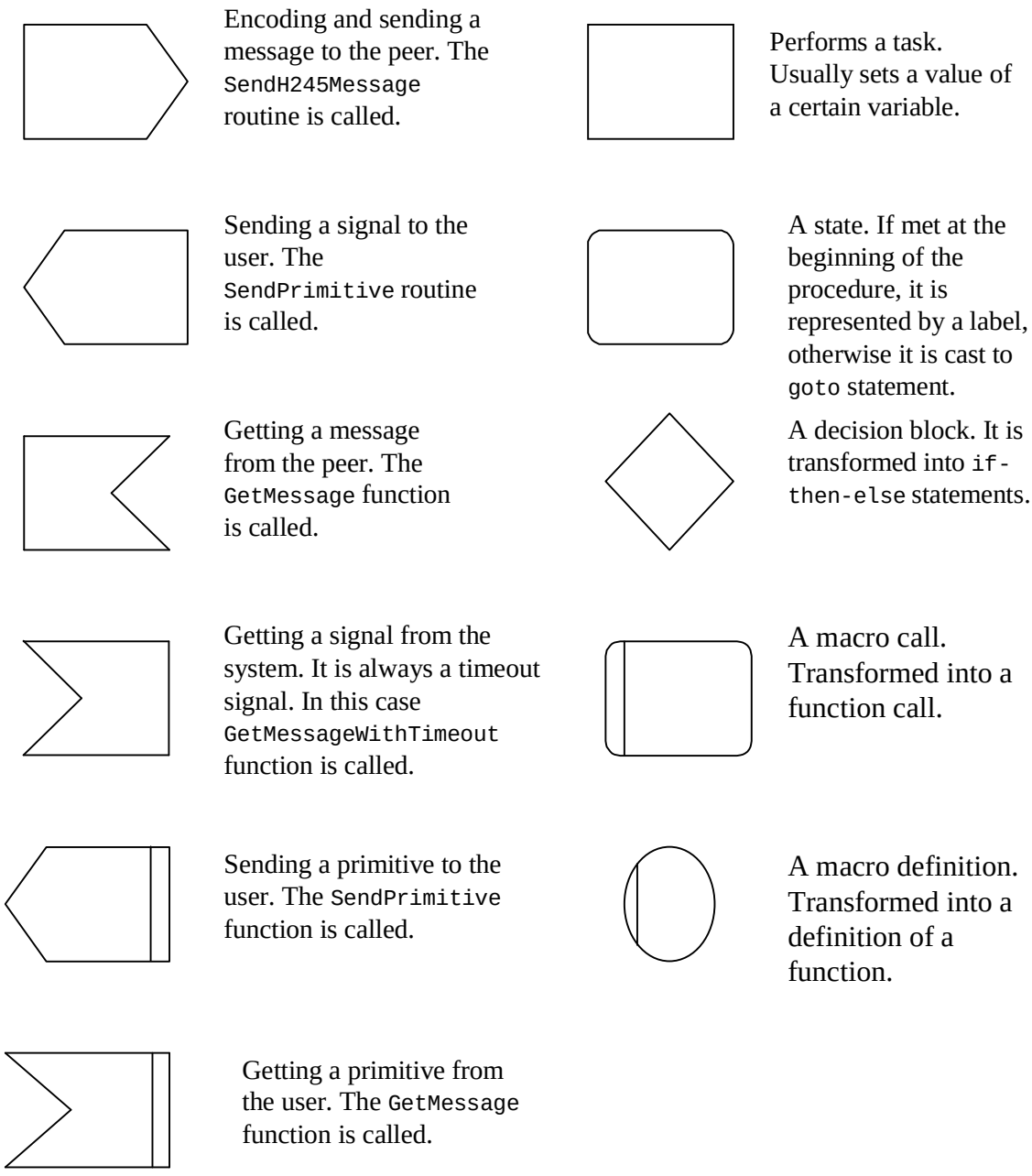

Figure 3-6. Conversion of SDL blocks into code.

3.3. Implementing primitives and messages

3.4. Programming Interface

The module, described here, contains all H.245 capabilities uses in H.323 terminal, i.e.: signalling entities - their creation, destruction and management; waiting for primitives (from the user) and messages (from the network); supports timers, error checking, separate commands and indications. The main idea of this module is to allow the user send all signals to the H.245 subsystem and forget about its internal structure. In this implementation just one H.245 can be run. So this module can be used as a part of a terminal (not gatekeeper, MCU, etc).

```
struct primitive_queue;
struct primitive_queue_node;
typedef struct primitive_queue *primitive_queue_p;
typedef struct primitive_queue_node *primitive_queuen_p;
```

Figure 3-7. Structures

Figure 3-7 shows what structures are used while communication with H.245 Subsystem is on. The mechanism used here is a set of queued primitives exchanged between the H.245 Subsystem and the Supervisory Thread. Type primitive_queue_p is a pointer to a queue where these primitives may be stored. It may be treated as a handler to a queue.

Full H.245 Subsystem API is presented in Figure 3-8.

```
primitive_queue_p create_primitive_queue();
int destroy_primitive_queue(primitive_queue_p queue_p);

int send_h245_primitive(int name, void *params);
int send_h245_primitive_p(struct primitive p);
struct primitive get_h245_primitive();
struct primitive get_h245_primitive_nb();

int get_primitive_name(struct primitive p);
void *get_primitive_parameter(struct primitive p);
```

Figure 3-8. H.245 Subsystem API

Before H.245 Subsystem can be started, some actions must be taken. First, a return queue must be created. The return queue is used by H.245 Subsystem to send primitives to its user. To create this queue the following `create_message_queue` function is used.

Next, there must be an opened socket. The socket must be passed directly to the H.245 Subsystem to allow it to transmit and receive messages through it. This socket is usually opened by the user of subsystem. A special type socket_struct has been prepared to make this module portable. Now, H.245 can be started.

To do that, the user calls the `start_H245` function. This function returns 0 if succeeds or non-zero if fails. The `stop_h245` function can be used to stop H.245 Subsystem: The function returns zero if succeeds or non-zero if fails (e.g. no subsystem was previously run).

The `send_h245_primitive` routine is used by the user to send primitives to H.245 Subsystem. *`mt`* is the name of the primitive. *`message_params`* is a pointer to parameters (if any). Primitives can be also sent using `send_h245_primitive_m`. In this case *`mesg`* is a manually propared message containing a primitive in the field *`mt`* and a parameter in *`mesg`*. Until now the following primitives have been defined : `H245_OPEN_LC` (open logical channel), `H245_CLOSE_LC` (close logical channel), `H245_MSD` (start master slave determination), `H245_CE` (start capability exchange), `H245_REQUEST_OPEN_LC` (request open logical channel), `H245_REQUEST_CLOSE_LC` (request close logical channel), `H245_REQUEST_CE` (request capability exchange), `H245_RTD` (start round trip delay procedures). Some of these primitives do not require parameters, e.g.:

```
send_h245_primitive(H245_MSD, NULL); /* Start MSD procedures */
```

The other require from user to fill a special structure (it differs for every primitive) and cast it on (void*) .

The `get_h245_primitive` function is used to acquire a primitive from the H.245 Subsystem. This function is a blocking one (if no parameters are in queue, it starts waiting until a parameter apppers). The function `get_h245_primitive_nb` is a non-blocking version of the `get_h245_primitive` routine (if no primitives are in the queue this function returns with EMPTY primitive). To get a primitive name `get_message_type` is used, and to get primitive parameter `get_message_parameter` is called.

Some more user-friendly API to this mechanism is being developed, e.g. `open_logical_channel(codec_id)`, `close_logical_channel(channel_num)`, ...

Bibliography

[1] *ITU-T Recommendation H.245: Control protocol for multimedia communication*, 07/97.

[2] *ITU-T Recommendation H.323: Packet-based multimedia communications systems*, 02/98.

[3] *ITU-T Recommendation X.680: Information Technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation*, 1994.

[4] *ITU-T Recommendation X.691: Information Technology - ASN.1 encoding rules - Specification of Packet Encoding Rules (PER)*, 1995.

[5] *ITU-T Recommendation H.225: Call signalling protocols and media stream packetization for packet based multimedia communication system*, 1998.

[6] *RFC 1889: RTP: A Transport Protocol for Real-Time Applications*, 1996, H. Schulzrinne, el al.

[7] *RFC 1890: RTP Profile for Audio and Video Conferences with Minimal Control*, January 1996, H. Schulzrinne.

[8] *A Primer on the H.323 Series Standard: http://www.databeam.com/h323/h323primer.html*, DataBeam Corp..

[9] *H-series Recommendations. Audiovisual and Multimedia Systems: http://www.itu.int/itudoc/itu-t/rec/h*, ITU-T.