Salutation Architecture Specification (Part-2)

Version 2.0c

June 01, 1999

Limitation of Liability

Even though the members of the Salutation Consortium have reviewed this Specification, the Consortium shall not make any warranty or representation, neither express or implied, with respect to this Specification, its quality or accuracy and it specifically disclaims the warranties of merchantability and fitness for a particular purpose.

No representation of third party rights

The Salutation Consortium makes no representation or warranty whatsoever with regard to the Consortium member or third party ownership, licensing or infringement/non-infringement of intellectual property rights. Each user of this Specification, whether or not a Consortium member, should seek the independent advice of legal counsel with regard to any possible violation of third party rights.

Trademarks

ESC/P (Epson Standard Code for Printers) is a trademark of EPSON Co.

IPDS (Intelligent Printer Data Stream) is a trademark of International Business Machines Corp.

LIPS (LBP Image Processing System) is a trademark of Canon Inc.

Microsoft Windows is a trademark of Microsoft Corporation.

NetWare is a trademark of Novell, Inc.

PAGES (Page Printer Advanced Graphics Escape Set) is a trademark of IBM Japan.

PCL (Printer Control Language) is a trademark of Hewlett-Packard Co.

PJL (Printer Job Language) is a trademark of Hewlett-Packard Co.

PostScript is a trademark of Adobe Systems Inc.

RPDL (Ricoh PDL) is a trademark of Ricoh Corp.

Sun RPC is a trademark of Sun Microsystems, Inc.

SCSA (Signal Computing System Architecture) is a trademark of Dialogic.

TSAPI (Telephony Services API) is a trademark of Novell, Inc.

Versit is a trademark of Apple Computer, Inc., International Business Machines Corp., Lucent Technologies, and Siemens Rolm Communications Inc.

Preface

The Part-1 of the Salutation Architecture Specification document defines the general framework of the architecture and the details of the Salutation Manager Protocol.

The Part-2, this document, consists of the following:

- The **Salutation Personality Protocol** of Functional Units, i.e. the format and protocol of messages, is defined in Chapters 1 through 4.
- The **Attributes** of each Functional Unit are defined. The Attributes definition is included in Chapters 2 through 4.
- Appendix (Chapters 5 and above) contains the definition of values, data types and syntax of all the defined protocol data units except [Fax Data] Functional Unit.

The Part-3 defines the criteria of the Conformance to the Salutation Architecture Specification.

Revision

Version 2.0 (December 02, 1996)

Public release of the final version 2.0 specification part 2.

Version 2.0a, 2.0b

No change and no release

Version 2.0c (June 01, 1999)

Added some attributes required for implementation and corrected the minor errors of the specifications.

Table of Contents

1.Common Framework

One of the characteristics of the Salutation Architecture is the provision of an common framework for service request commands and protocols which is independent of the type of service and data format. The common framework provides the following advantages:

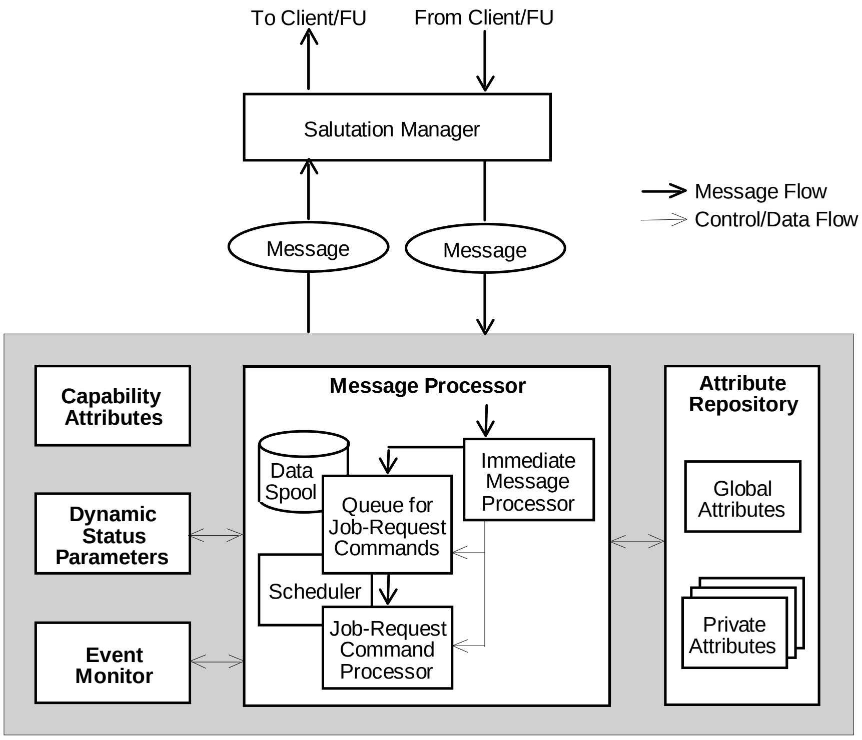
- Easy to learn interface specifications
- Easy to implement coordinated functions
- Easy to expand architecture (easy to add new services)

This chapter describes the characteristics of **Salutation Personality Protocol** that are common across Functional Units.

1.1.Virtual Model of Functional Unit under Salutation Personality Protocol

A Functional Unit exchanges messages with a client, which is either a Salutation client application or another Functional Unit, through a Service Session. When a Service Session is opened by *Open Service*, the protocol to be used in the session is specified by the Personality Protocol ID parameter in the *Open Service* request. If the Salutation Personality Protocol is specified at *Open Service*, messages exchanged between the client application and the Functional Unit or between the two Functional Units in the session follow the definition of this part-2 of the architecture specification document.

The following figure shows the virtual model of a Functional Unit under the Salutation Personality Protocol.



Functional Unit (under Salutation Personality Protocol)

- Capability Attributes

  The Capability Attributes describe the detail of services the Functional Unit can provide. When the Functional Unit registers its capability to the Salutation Manager, it gives the Salutation Manager a Functional Unit Description Record that contains the Capability Attributes. The values of Capability Attributes are static and do not change. A client may query the values of Capability Attributes by issuing a *Query Capability*.

- Dynamic Status Parameters

  Each Dynamic Status Parameter describes an aspect of the current Functional Unit status. The values of Dynamic Status Parameters are dynamic and change. A client may query the current value of a particular Dynamic Status Parameter by sending a QueryDynamicStatus command.

- Event Monitor

  A client may request the Functional Unit to notify the client of any changes of a particular Dynamic Status Parameter. Making such a request is called "subscribing to an event". The Event Monitor monitors the values of subscribed Dynamic Status Parameters, and generates an "event" when the value changes. The "event" is notified to the client by a NotifyEvent message.

- Message Processor

  The Message Processor receives all the messages for the Functional Unit. It processes message by message, and if necessary, builds a message to be sent back to the client.

It is also notified of the initiation and termination of Service Sessions (*Open Service ~ Close Service*), and performs session-related housekeeping tasks as required.

Depending on the received messages, the Message Processor operates on the Attribute Repository.

Some messages that take relatively long time to process are not executed immediately. For example, if a job-request-type command (see "Job-Related Messages" section on page 33) is received, the Message Processor performs the following tasks:

1) Creates a job instance
2) Assigns a JobHandle
3) Builds a response to return the JobHandle to the client. It is sent to the client either at this step or at the end of the next step depending on the received command.
4) If the job-request-type command indicates that associated data item(s) must be obtained immediately, the Message Processor initiates Data Transfer Message Sequence (see “Data Transfer Messages” section on page 18) to get the data item(s). (Note that data transfer occurs either when the job-request-type command is received as described here, or when the job-request-type command is actually executed, depending on the command or its parameters.)
5) Enqueues the job-request-type command and, if present, the associated data item(s) in the Job Queue and the Data Spool respectively. If necessary, default attribute values are assigned to the command from the attributes in the Attribute Repository at this stage. See the description of Attribute Repository below for the detail.

The queued commands will be processed by the Queued Job-Request Command Processor in turn. The Scheduler determines which queued command is to be processed next, by inspecting the queued commands. Some Functional Units do not require, or allow optional implementation of the Queue and the Data Spool capability.

- Attribute Repository

An **attribute** in the Attribute Repository is a parameter which controls a way services are done by the Functional Unit. It is sometimes called "**Command Attribute**" to discriminate it from a Capability Attribute. The specification of each Functional Unit defines all the attributes associated with the Functional Unit.

The Attribute Repository contains the following two types of attributes.

☐ **Global Attribute**

A single image of the Global Attributes is shared by all the clients of the Functional Unit.

Global Attributes are read only.

The value of each Global Attribute is implementation dependent.

A typical use of Global Attribute is to set default parameter values for the equipment by the administrator.

☐ **Private Attribute**

Private Attributes are private to a client, and not shared by the other clients. A separate image of the Private Attributes is associated with each client of the Functional Unit.

Private Attributes are settable (read/write).

Initially, there is no Private Attribute in the Attribute Repository.

A typical use of Private Attribute is to set default parameter values that are applied to subsequent commands.

Private Attributes values are valid only until the current Service Session (*Open Service ~ Close Service*) is terminated.

The proper use of Global Attribute and Private Attribute can provide a flexible way of specifying an attribute value, as follows:

1) The attribute value specified in the command is used, if present.
2) Otherwise, the attribute value specified in the Private Attribute is used, if present.
3) Otherwise, the attribute value specified in the Global Attribute is used. (If the attribute is not defined as a Global Attribute, the architecture-defined default value is used. An example is the “printCopyCount” attribute of the [Print] Functional Unit.)

In the cases of 2) or 3), the default attribute value is assigned from the Private Attribute or the Global Attribute respectively to each command when the command is received by the Functional Unit. It is NOT that the value of the Private/Global Attribute is used at the time the command is executed. Therefore, in the following example, the *COMMAND* uses the value "aaa" for XYZ attribute.

Client	**Server**
SetPrivateAttribute({XYZ="aaa"}) =>	
	<= ACK(NULL)
COMMAND(XYZ attribute is omitted) =>	
	The server enqueues the *COMMAND*.
:	
SetPrivateAttribute({XYZ="bbb"}) =>	
	<= ACK(NULL)
:	
	The server dequeues the *COMMAND* and execute it. (XYZ="aaa" is used.)

1.2.Message

The Functional Unit receives a command from a client application or another Functional Unit (called the client hereafter), and sends a response to the client. In some cases, the Functional Unit sends a command to the client which returns a response. In the rest of this document, a **message** is defined as either a command or a response.

There are common commands and responses, which are commonly used across the Functional Units, and there are Functional Unit specific commands used in the Functional Unit. What kind of common commands and responses are supported, and what kind of Functional Unit specific commands are defined for the Functional Unit is described in each chapter of the Functional Unit. Mandatory support or optional support by the Functional Unit is also defined in each description of the Functional Unit. If commands are categorized mandatory support, Functional Unit must support these commands and recognize them for further handling, however support of optional commands depends on the Functional Unit implementation. The Functional Unit will inform a client what Optional support commands are supported by the supportedCommand attribute in the Capability Attribute.

The command consists of command itself and parameters to control the services in the Functional Unit. Some parameters are defined as optional, therefore Functional Unit does not need to support all of the parameters defined in the specification. The command sender can set the optional parameters if the Functional Unit supports them.

1.3.Message Sequence

A **message sequence** is the sequence of two or more related messages exchanged between the client and the Functional Unit alternately. Some message sequences are initiated by the client, and the other message sequences are initiated by the Functional Unit.

A complete message sequence consists of one command and one response which is either **ACK** or **NACK** except for data transfer message sequence which is described in "Data Transfer Messages" section on page 18.

A message sequence is initiated by a command. Every message has a parameter called MsgSeqID in its header. The MsgSeqID field is INTEGER data type. It is used as follows:

- When the client application or Functional Unit that has initiated the service session by sending an Open Service request is going to start a new message sequence, it assigns a positive value to the MsgSeqID field of the initiating command. All the rest of commands and responses in this message sequence have the same value in their MsgSeqID field.

  The MsgSeqID value is increased by one for each new message sequence. It wraps back to 1 after a sufficiently large number, e.g. 32767 (‘7FFF’ in hexadecimal).

- When the Functional Unit that has accepted the Open Service request is going to start a new message sequence, the same rule is followed except that a negative value is assigned in the MsgSeqID field.

  The MsgSeqID value is decreased by one for each new message sequence. It wraps back to -1 after a sufficiently small number, e.g. -32768 (‘8000’ in hexadecimal).

A message sequence is terminated by and only by ACK, NACK, or the closing of the service session.

When the client or the Functional Unit (FU) initiates a new message sequence, it must follow the following rules:

- The client/FU must not initiate a new message sequence while another message sequence it has initiated is not terminated.

Client	**Server**
-- start of message sequence-a --	
COMMAND (MsgSeqID=1) =>	
	<= *COMMAND* (MsgSeqID=1)
COMMAND (MsgSeqID=1) =>	
	<= ACK or NACK (MsgSeqID=1)
-- end of message sequence-a --	
-- start of message sequence-b --	
COMMAND (MsgSeqID=2) =>	
	<= *COMMAND* (MsgSeqID=2)
ACK or NACK (MsgSeqID=2)=>	
-- end of message sequence-b --	
-- start of message sequence-c --	
COMMAND (MsgSeqID=3) =>	
	<= ACK or NACK (MsgSeqID=3)
-- end of message sequence-c --	
:	

- The client/FU may initiate a new message sequence even if there is an on-going message sequence provided that the on-going message sequence has been initiated not by the client/FU but by the other end of the service session.

Client	**Server**
-- start of message sequence-a --	
COMMAND (MsgSeqID=1) =>	
-- start of message sequence-b --	
	<= *COMMAND* (MsgSeqID=-1)
ACK or NACK (MsgSeqID=-1)=>	
-- end of message sequence-b --	
	<= ACK or NACK (MsgSeqID=1)
-- end of message sequence-a --	

One whole Transfer Data packet contains one and only one message under the Salutation Personality Protocol.

1.4.Common Messages

The following messages are commonly used across Functional Units under the Salutation Personality Protocol as a part of the common framework. The specification of each Functional Unit dictates which message is actually supported and in what context it is used.

1.4.1.ACK, NACK Messages

These are the simplest forms of common response against various commands. *ACK* positively acknowledges the previous command, and *NACK* negatively acknowledges it.

ACK takes zero or more parameters, of which data type or meaning depend on the associated command. In the rest of this document, the following convention is used to describe parameters of *ACK* message.

- ACK(NULL) : ACK takes no parameter
- ACK(*type*) : ACK takes one parameter whose data type is *type*.
- ACK(*type-1, type-2*) : ACK takes two parameters. The data type of the first parameter is *type-1*, that of the second is *type-2*.
- (and so on)

NACK takes one mandatory parameter value, **ReturnCode**, that describes why the previous command is rejected or failed to be processed. *NACK* may take an optional parameter to describe the detail or additional information regarding the reason of rejection/failure. This optional parameter is mainly intended for implementation convenience (e.g. for debug or diagnostics), and may be ignored by the receiver of *NACK* response. The existence and content of this optional parameter is entirely left to each implementation.

How *ACK/NACK* is used is described in the description of each associated command.

ASN.1 Syntax Definition

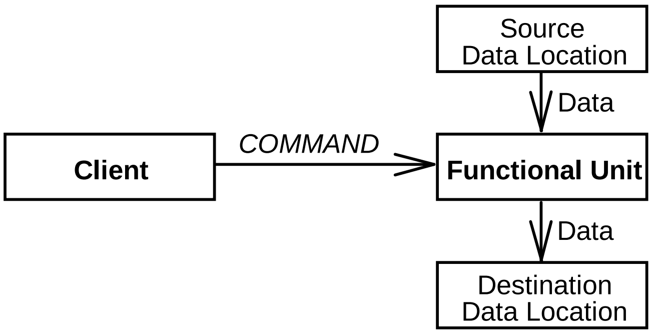
```
ACK                          ::= [APPLICATION tagACK] SEQUENCE
{
                                 COMPONENTS OF MsgHeader,
   parameter1                    [0] ANY OPTIONAL,
   parameter2                    [1] ANY OPTIONAL,
   parameter3                    [2] ANY OPTIONAL
   --  :                               :
   -- The number and data type of parameters depend on the associated command, and are defined
   -- by the specification of each associated command.
}
```

```
NACK     ::= [APPLICATION tagNACK] SEQUENCE
{
                                        COMPONENTS OF MsgHeader,
    returnCode                          [0] ReturnCode,
    descriptor                          [1] OCTET STRING        OPTIONAL
                                            -- Additional information for the reason of rejection.
                                            -- Debug/diagnostics purpose. May be ignored.
}
```

1.4.2.Data Transfer Messages

1.4.2.1.Overview

"Data" represents a meaningful unit of data sequence, such as compound document, file, etc., which is treated in a certain consistent manner by client/server applications with the Salutation Personality Protocol.



A data transfer is initiated by a *COMMAND* that is sent by a client application or another Functional Unit (the client) to a Functional Unit. Depending on the *COMMAND*, the data is transferred into the Functional Unit and/or the data is transferred out of the Functional Unit.

The *COMMAND* specifies:

- how the "data" should be processed, and
- where the "data" to be processed is (source data location) and/or where the processed "data" should be (destination data location)

Some examples of such *COMMAND* are Print (data transfer into FU), RetrieveDoc (data transfer out of FU), and commands for data format conversion (data transfer into and out of FU).

1.4.2.1.1.Data Transfer Mode

When the *COMMAND* initiates data transfer **into the Functional Unit**, the following two **Data Transfer Mode**s are defined. If the *COMMAND* is a job-request-type (refer to “Job-Related Messages” section on page 33), the *COMMAND* explicitly or implicitly specifies the mode to be used. If the *COMMAND* is not a job-request-type, data transfer is always in immediate mode.

- **Immediate Mode**

Data transfer is initiated immediately after the *COMMAND* is received by the Functional Unit (before the *COMMAND* is queued in the job queue if it is a job-request-type.)

- **Delayed Mode**

  Data transfer is initiated when the *COMMAND* is executed by the Functional Unit, after the *COMMAND* is dequeued from the job queue.

The delayed-mode data transfer is useful when the Functional Unit does not have a large storage to pool data.

1.4.2.1.2.Data Location

The source/destination data location is specified as one of the following:

- **Client**

  The source/destination data location is the client itself that is issuing the *COMMAND* to initiate the data transfer.

- **Export Pool**

  This choice may be used only as the destination data location.

  When it is specified, the “data” is not actually transferred but is prepared for another Functional Unit to access.

- **Functional Unit**

  This choice may be used only as the source data location.

  The data is transferred from the specified Functional Unit (data-source FU). The data-source FU is identified by the Functional Unit Handle of the data-source FU and the SLM-ID of the SLM with which the data-source FU is registered.

- **URL**

It is optional to support this choice. The Functional Unit indicates in its capability attribute whether or not it supports URL-based data location specification.

The data is transferred from/to a file system designated by the specified Uniform Resource Locator (URL). Only the URL with either “ftp” or “file” scheme is allowed. If the “ftp” scheme is used, the Functional Unit accesses the specified file under the File Transfer Protocol (FTP). If the “file” scheme is used, the Functional Unit accesses the specified file in an implementation dependent way.

Unlike the other three choices, if the URL-based data location is specified, it is not possible to transfer the description (format) of the “data” together with the “data” content or to retain the data block boundaries (described in “Data Transfer Message Sequence” section below). The URL-based data location should not be used when the data format is not obvious from the file name or when the data block boundaries need to be maintained.

The rest of “Data Transfer Messages” section defines the framework for the data transfer between a client application and a Functional Unit, or between a Functional Unit and another Functional Unit, and is not applicable to data transfer to/from the URL-specified data location.

1.4.2.1.3.DataHandle

DataHandle is used to identify the "data" to be transferred. DataHandle is always assigned by the sender of the data. Data-sending FU implementation must generate a sufficiently long and random value for a DataHandle.

A DataHandle is valid only for one Data Transfer Message Sequence. After the completion of the Data Transfer Message Sequence, the sender invalidates the DataHandle. If the same or different receiver attempts to request another data transfer using the same DataHandle, the sender rejects it by "Unknown DataHandle" error.

When a client application requests a Functional Unit (data-source FU) to send data to another Functional Unit (data-destination FU), the client application sets the data destination parameter in the *COMMAND* to the data-source FU as the **Export Pool**. The data-source FU generates a DataHandle and returns it to the client application. Such data handle is called **Export DataHandle**. The client application then sends another *COMMAND* with the Export DataHandle to the data-destination FU. The data-destination FU sends a *RequestDataTransfer* command (described in the next section) with the Export DataHandle to the data-source FU to initiate receiving the data.

The Export DataHandle is not invalidated even if the client application closes the service session after it receives the Export DataHandle from the data-source FU. There may be cases that the data source FU has assigned an Export DataHandle, but no receiver will have requested a data transfer with the Export DataHandle. Each implementation may define when unused Export DataHandles are invalidated, for example, once a day, or limiting the number of pending handles to twenty.

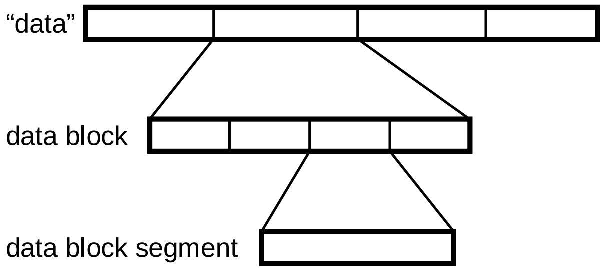
1.4.2.1.4.Data Transfer Message Sequence

A **Data Transfer Message Sequence** is used to transfer the data between the source/destination data location and the Functional Unit. It consists of the following messages:

- RequestDataTransfer
- DataBlockDescription
- TransferDataBlock
- RequestNextData
- ACK
- NACK

In this section, a "data" is defined as the whole of data to be transferred in one complete Data Transfer Message Sequence.

A "data" consists of one or more data blocks. One data block consists of one or more data block segments. One data block segment is transferred by one *TransferDataBlock* message. The message has two flags (Begin Data Block and End Data Block) to indicate data block boundaries and an additional flag (Last Segment) to indicate the last segment of the last data block.



It is allowed to mix different format (or any other optional attributes) of data blocks in a "data". (However, one data block shall not contain mixed formats). *DataBlockDescription* message is used to specify the format of subsequent data blocks. It may be omitted If the format of data block is known to the receiver, for example, because the receiver and the sender have agreed on the format by a separate message in advance, or because the specification of Functional Unit has defined the default format.

A "data" is transferred in one data block unless it contains mixed formats or the specification of each Functional Unit specifies when a "data" should be divided into multiple data blocks. For example, when a document in bi-level image stream format is printed by a [Print] Functional Unit under the Salutation Personality Protocol, the document “data” is transferred to the FU in multiple data blocks, each data block corresponding to a page in the document.

On the other hand, it is an implementation matter how to divide each data block into multiple data block segments. For example, if a particular SLM implementation imposes the maximum size for the application data parameter of *slmTransferData()* SLM-API, the sending application may have to split each data block so that each data block segment fits in the limited size.

The following is the simplest form of Data Transfer Message Sequence.

Sender	**Receiver**
TransferDataBlock(Begin, End, Last) =>	
	<= ACK

The following is more complex example of Data Transfer Message Sequence. In this example, the "data" consists of three data blocks, the first two in one format, and the third in another format. The second data block is further divided into two data block segments.

Sender		Receiver
		<= RequestDataTransfer(DataHandle)
	data format for the first & second data block	
DataBlockDescription =>		
		<= RequestNextData
	the first data block	
TransferDataBlock(Begin, End) =>		
		<= RequestNextData
	the second data block	
TransferDataBlock(Begin) =>		
		<= RequestNextData
TransferDataBlock(End) =>		
		<= RequestNextData
	data format for the third data block	
DataBlockDescription =>		
		<= RequestNextData
	the third data block	
TransferDataBlock(Begin, End, Last) =>		
		<= ACK

The same MsgSeqID value is used in all messages in a Data Transfer Message Sequence.

1.4.2.1.5.Simplified Data Transfer Message Sequence

When a *COMMAND* initiates the transfer of **only one "data" from the client** to the Functional Unit under **immediate mode**, the Functional Unit omits the ACK response to the *COMMAND* and immediately sends a RequestDataTransfer message.

When a *COMMAND* initiates the transfer of **only one "data"** from the Functional Unit **to the client**, the Functional Unit omits both ACK response and RequestDataTransfer message.

In both cases, DataHandle is not used.

The same MsgSeqID value is used in all messages from the initiating *COMMAND* through the completion of the data transfer in a simplified data transfer message sequence.

1.4.2.1.6.Usage of Data Transfer Message Sequence

The following examples show typical uses of Data Transfer Message Sequence. In the examples, the shortest possible Data Transfer Message Sequence is shown only for the sake of simplicity, however the "data" may actually consist of multiple data blocks, each data block consisting of multiple data block segments.

1.4.2.1.6.1.Data Transfer from Client to Functional Unit : Delayed Mode

The client issues a *COMMAND* with a DataHandle that identifies the client's data to be transferred to the Functional Unit. The *COMMAND* may be queued and not processed immediately by the Functional Unit. The Functional Unit requests the client to transfer the data when the *COMMAND* is dequeued and executed.

Client **Server**

--- Message Sequence Start (MsgSeqID=n1)---

COMMAND(DataHandle, ...) =>

<= ACK(...)

--- Message Sequence End (MsgSeqID=n1)---

COMMAND may be queued.

--- Message Sequence Start (MsgSeqID=n2)---

<= RequestDataTransfer(DataHandle)

TransferDataBlock(Begin, End, Last) =>

<= ACK(NULL)

--- Message Sequence End (MsgSeqID=n2)---

If more than one data item is to be transferred, the Functional Unit initiates a separate Data Transfer Message Sequence for each data item, as follows.

Client	**Server**
--- Message Sequence Start (MsgSeqID=n1)---	
COMMAND(DataHandle-1, DataHandle-2, ...) =>	
	<= ACK(...)
--- Message Sequence End (MsgSeqID=n1)---	
COMMAND may be queued.	
--- Message Sequence Start (MsgSeqID=n2)---	
	<= RequestDataTransfer(DataHandle-1)
TransferDataBlock(Begin, End, Last) =>	
	<= ACK(NULL)
--- Message Sequence End (MsgSeqID=n2)---	
--- Message Sequence Start (MsgSeqID=n3)---	
	<= RequestDataTransfer(DataHandle-2)
TransferDataBlock(Begin, End, Last) =>	
	<= ACK(NULL)
--- Message Sequence End (MsgSeqID=n3)---	

1.4.2.1.6.2.Data Transfer from Client to Functional Unit : Immediate Mode

If more than one data item is to be transferred, the flow is the same as above except that data items are transferred before the *COMMAND* is queued.

If only one data item is to be transferred, simplified data transfer message sequence is used as follows:

Client	**Server**
--- Message Sequence Start (MsgSeqID=n1)---	
COMMAND(...) =>	
	<= RequestDataTransfer()
TransferDataBlock(Begin, End, Last) =>	
	<= ACK(...)
--- Message Sequence End (MsgSeqID=n1)---	

1.4.2.1.6.3.Data Transfer from Functional Unit to Client

If more than one data item is to be transferred, the client application initiates a separate Data Transfer Message Sequence for each data item, as follows.

Client	**Server**

--- Message Sequence Start (MsgSeqID=n1)---

COMMAND(...) =>

<= ACK(DataHandle-1, DataHandle-2, ...)

--- Message Sequence End (MsgSeqID=n1)---

--- Message Sequence Start (MsgSeqID=n2)---

RequestDataTransfer(DataHandle-1) =>

<= TransferDataBlock(Begin, End, Last)

ACK(NULL) =>

--- Message Sequence End (MsgSeqID=n2)---

--- Message Sequence Start (MsgSeqID=n3)---

RequestDataTransfer(DataHandle-2) =>

<= TransferDataBlock(Begin, End, Last)

ACK(NULL) =>

--- Message Sequence End (MsgSeqID=n3)---

If only one data item is to be transferred, simplified data transfer message sequence is used as follows:

Client	**Server**

--- Message Sequence Start (MsgSeqID=n1)---

COMMAND(...) =>

<= TransferDataBlock(Begin, End, Last)

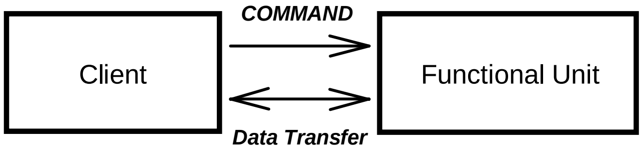
ACK(NULL) =>

--- Message Sequence End (MsgSeqID=n1)---

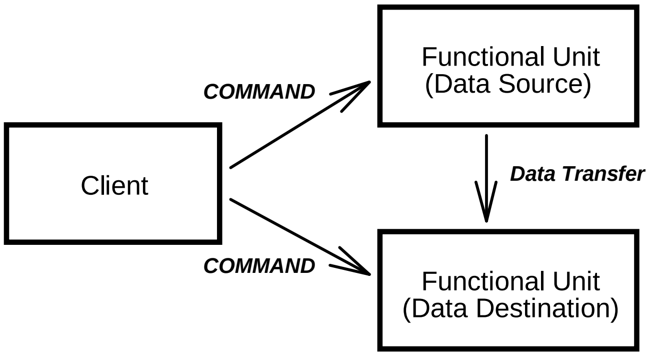
1.4.2.1.6.4.Data Transfer between Functional Units

Examples so far have shown data transfers between the client and the Functional Unit. A client application can also request a data transfer to occur between two Functional Units.

Data Transfer between Client and Functional Unit



Data Transfer between Functional Units



The procedure for a data transfer between Functional Units is as follows:

1) The client establishes a service session with the data-source Functional Unit.

2) The client sends a *COMMAND* to the data-source FU specifying that the destination of data transfer is "Export Pool" (not the client). When the Export Pool is selected as the destination, the data is not actually transferred but is just prepared for another Functional Unit to access. The data-source FU assigns an Export DataHandle for the data item to be transferred, and returns it to the client. (The session between the client and the data-source FU may be closed hereafter.)

3) The client establishes a service session with the data-destination Functional Unit.

4) The client sends a *COMMAND* to the data-destination FU to give the Export DataHandle together with **AbsoluteFunctionalUnitHandle**. AbsoluteFunctionalUnitHandle contains the following information of the data-source FU:

   - The SLM-ID of the SLM with which the data-source FU is registered
   - The Functional Unit Handle of the data-source FU

   (The session between the client and the data-destination FU may be closed hereafter.)

5) The data-destination Functional Unit establishes a service session with the data-source Functional Unit.

6) The data-destination FU initiates the data transfer message sequence by sending a RequestDataTransfer message with the Export DataHandle.

7) The data-destination Functional Unit closes the session with the data-source Functional Unit after the data transfer is completed.

The following figure shows how messages are actually exchanged.

Client	**Data-Source FU**

Client establishes a session with data-source Functional Unit.

--- Message Sequence Start (MsgSeqID=n1)---

COMMAND(DataDestination=ExportPool, ...) =>

<= ACK(DataHandle)

--- Message Sequence End (MsgSeqID=n1)---

Client	**Data-Destination FU**

Client establishes a session with data-destination Functional Unit.

--- Message Sequence Start (MsgSeqID=n2)---

COMMAND(DataSource=AbsoluteFunctionalUnitHandle, DataHandle, ...) =>

<= ACK(...)

--- Message Sequence End (MsgSeqID=n2)---

Data-Destination FU	**Data-Source FU**

Session is established between the Functional Units.

--- Message Sequence Start (MsgSeqID=n3)---

RequestDataTransfer(DataHandle) =>

<= TransferDataBlock(Begin, End, Last)

ACK(NULL) =>

--- Message Sequence End (MsgSeqID=n3)---

1.4.2.1.7.Skeleton of Data-Transfer-Initiating Command

Some of the parameters shown in the following skeleton may be defined as optional or not included in each data-transfer-initiating *COMMAND* definition.

```
COMMAND                  ::= [APPLICATION tagCOMMAND] SEQUENCE
{
                             COMPONENTS OF MsgHeader,
     : (other parameters)
   modeOfDataTransfer        [ ] DataTransferMode,
   dataSource                [ ] DataLocation         DEFAULT client,
   dataHandle                [ ] DataHandle,
   dataDestination           [ ] DataLocation         DEFAULT client,
     : (other parameters)
}
```

1.4.2.2.Message Description

1.4.2.2.1.RequestDataTransfer

The receiver of the data initiates the Data Transfer Message Sequence by this message. It identifies the "data" to be transferred by **DataHandle** parameter. DataHandle must have been assigned by the sender and known to the receiver by a separate message in advance, as described in the previous "Usage of Data Transfer Message Sequence" section on page 22.

The DataHandle parameter in RequestDataTransfer message is omitted in simplified data transfer message sequence.

Response

One of the following is returned in response to this message:

- *DataBlockDescription*, *TransferDataBlock*

  The sender has accepted the request and started the data transfer.

- *NACK*

  The sender has failed to process the message. *NACK* includes a **ReturnCode** which indicates the reason of the failure. *NACK* terminates the message sequence.

ASN.1 Syntax Definition

```
RequestDataTransfer      ::= [APPLICATION tagRequestDataTransfer] SEQUENCE
{
                             COMPONENTS OF MsgHeader,
   dataHandle                [0] DataHandle OPTIONAL
}
```

1.4.2.2.2.DataBlockDescription

This message tells the receiver of the format and/or other attributes of subsequent data blocks in this Data Transfer Message Sequence.

The specification of each Functional Unit dictates whether this message is used or not, when this message is used, and what parameter this message takes.

Response

One of the following is returned in response to this message:

- *RequestNextData*

  The receiver has processed the message successfully and is ready to receive next message.

- *NACK*

  The receiver has failed to process the message. *NACK* includes a **ReturnCode** which indicates the reason of the failure. *NACK* terminates the message sequence.

ASN.1 Syntax Definition

```
DataBlockDescription            ::= [APPLICATION tagDataBlockDescription] SEQUENCE
{
                                    COMPONENTS OF MsgHeader,
    dataDescriptor                  [0] CHOICE
    {
        document                    [0] DocumentDataDescriptor,
        file                        [1] FileData
    }
}
```

1.4.2.2.3.TransferDataBlock

This message is used by the sender to transfer a data block segment to the receiver. A separate *TransferDataBlock* message is used for each data block segment.

It has flags to indicate the first and last data block segment in a data block. If a data block consists of only one data block segment, both flags are set and entire data block is transferred by one *TransferDataBlock* message.

This message also has another flag to indicate the last data block segment of the last data block. For example, if the entire “data” is transferred by one *TransferDataBlock* message, all the three flags are set.

Note: “TransferDataBlock” message should not be confused with “*Transfer Data*” RPC message of the Salutation Manager Protocol.

Response

One of the following is returned in response to this message:

- *RequestNextData*

  The receiver has processed the message successfully and is ready to receive next message.

- *ACK*

  This response is returned only if the "last data block segment" flag is set in the preceding *TransferDataBlock* message, and the receiver has successfully processed the message. This is the last message of a successful Data Transfer Message Sequence.

  Unless otherwise specified by the specification of each Functional Unit, *ACK* takes no parameter.

- *NACK*

  The receiver has failed to process the message. *NACK* includes a **ReturnCode** which indicates the reason of the failure. *NACK* terminates the message sequence.

ASN.1 Syntax Definition

```
TransferDataBlock               ::= [APPLICATION tagTransferDataBlock] SEQUENCE
{
                                    COMPONENTS OF MsgHeader,
    beginDataBlock                  [0] BOOLEAN,
    endDataBlock                    [1] BOOLEAN,
    lastSegment                     [2] BOOLEAN,  -- TRUE in the last data block segment of the
                                                  -- last data block of “data”
    dataBlockBody                   [3] OCTET STRING
}
```

1.4.2.2.4.RequestNextData

The receiver issues this message when it has successfully processed the previous message from the sender, and when it is ready to receive next message.

Response

One of the following is returned in response to this message:

- *DataBlockDescription*, *TransferDataBlock*

  The sender is continuing the data transfer.

- *NACK*

  The sender has failed to process the message. *NACK* includes a **ReturnCode** which indicates the reason of the failure. *NACK* terminates the message sequence.

ASN.1 Syntax Definition

```
RequestNextData                 ::= [APPLICATION tagRequestNextData] SEQUENCE
{
                                    COMPONENTS OF MsgHeader
}
```

1.4.3.Attribute Repository Messages

1.4.3.1.Overview

The following messages are defined to read/set values of attributes in the Attribute Repository. Each message may include more than one attribute.

- GetPrivateAttribute
- GetGlobalAttribute
- SetPrivateAttribute

All of these messages are not mandatory support for the Functional Unit. Supported message tags by an FU will be set in the supportedCommand attribute in the capability attributes.

The normal flow of these commands and responses are as follows:

Client	**Server**

To read attribute values:

GetPrivateAttribute(SET OF AttributeID) =>
or
GetGlobalAttribute(SET OF AttributeID) =>

<= ACK(AttributeList)

To set attribute values:

SetPrivateAttribute(AttributeList) =>

<= ACK(NULL)

The initial version of the architecture has not defined SetGlobalAttribute message.

1.4.3.2.Message Description

1.4.3.2.1.GetPrivateAttribute, GetGlobalAttribute

The client sends a *GetPrivateAttribute* or a *GetGlobalAttribute* message to read the value of one or more Private Attributes or Global Attributes respectively. The message has a set of Attribute IDs as its parameter.

The server returns the attribute ID-value pairs of only those attributes that exist in the Attribute Repository. The attributes that do not exist, either because they are not supported or because their values have not been set by SetPrivateAttribute message, are not included in the response.

Response

One of the following is returned in response to this message:

- *ACK*(AttributeList)

  The server is returning the valid ID-value pairs of queried attributes.

- *NACK*(ReturnCode)

  The server has failed to process the command. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
GetPrivateAttribute          ::= [APPLICATION tagGetPrivateAttribute] SEQUENCE
{
                                 COMPONENTS OF MsgHeader,
    attributeIdList              [0] SET OF AttributeID
}
```

```
GetGlobalAttribute              ::= [APPLICATION tagGetGlobalAttribute] SEQUENCE
{
                                    COMPONENTS OF MsgHeader,
    attributeIdList                 [0] SET OF AttributeID
}

AttributeList                   ::= SET OF SEQUENCE
{
    attributeId                     [0] AttributeID,
    attributeValue                  [1] ANY          -- Type is defined by each attribute.
}
```

1.4.3.2.2.SetPrivateAttribute

The client sends a *SetPrivateAttribute* message to set the value of one or more Private Attributes respectively. The message has a set of Attribute ID-value pairs as its parameter.

If any of the attributes in the message is unknown or not supported by the server, the entire message is rejected and no attribute value is changed.

If the value parameter is omitted, the corresponding attribute in the Attribute Repository is deleted. The command is accepted even if the attribute to be removed does not exist in the Attribute Repository provided that the attribute is supported.

Response

One of the following is returned in response to this message:

- *ACK*(NULL)

  The server has successfully set attribute values.

- *NACK*(ReturnCode)

  The server has failed to set attribute values. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
SetPrivateAttribute             ::= [APPLICATION tagSetPrivateAttribute] SEQUENCE
{
                                    COMPONENTS OF MsgHeader,
    attributeList                   [0] SET OF SEQUENCE
    {
        attributeId                 [0] AttributeID,
        attributeValue              [1] ANY          OPTIONAL
                                        -- Type is defined by each attribute.
                                        -- If omitted, attribute is deleted.
    }
}
```

1.4.4.Job-Related Messages

1.4.4.1.Overview

1.4.4.1.1.Job-Request-Type Command and Job Entry

For a command that may take time to be processed, the server assigns and returns a **JobHandle** to the client in *ACK* response as follows, when the command is received but before the command is executed.

Client	**Server**
COMMAND =>	
	<= ACK(JobHandle)
	(The COMMAND is enqueued in the job queue.)
	:
	(The COMMAND is dequeued from the job queue and executed.)

This type of command is called **Job-Request-Type** command. The client may send a *CancelJob* message to the server to cancel the associated job-request-type command. The server deletes the command if it is in a queue and not processed yet, or optionally abort the command execution if it is being processed. See the description of *CancelJob* message for more detail.

Client	**Server**
CancelJob(JobHandle) =>	
	<= ACK(NULL)

Certain job-request-type commands are structured such that a job consists of more than one task. For example, a command to send a document to multiple destinations by FAX consists of multiple tasks. Each task sends a document to one named destination. Each task in a job-request-type command is called a **Job Entry**. The client assigns a unique integer value, called **JobEntryID**, to each Job Entry. Note that JobHandle is assigned by the server but JobEntryID is assigned by the client. JobEntryID can be used, for example, to know the result of each Job Entry execution individually later.

Job or Job Entry related messages are not mandatory support for the Functional Unit. Supported message tags by an FU will be set in supportedCommand attribute in the capability attributes.

1.4.4.1.2.Life of Job

JobHandle, JobEntryID, and the status of job execution have one of the following life. The server must retain these values, and the client may refer to JobHandle and JobEntryID in messages, only during their life.

- Job (default)

  The life terminates after the server sends the result of job execution to the client at the completion of the job, or if there is no notification to be sent, the life terminates at the completion of job execution. The server must retain JobEntryID and status of all job entries until all job entries are executed.

Note that the life is NOT terminated by the termination of current Service Session (*Open Service ~ Close Service*).

- Session

  The life terminates at the end of current Service Session (*Open Service ~ Close Service*). It may be terminated also by *FreeJobHandle* message from the client.

  If the job is still queued in the job queue or it is being executed when the current session is terminated, the job is purged from the job queue (together with any spooled data, if any) or the job execution is aborted respectively.

  The server is desirable to retain JobEntryID and status of all job entries until current session is terminated or JobHandle is freed.

- Persistent

  The life is not terminated by the end of current Service Session or by the completion of the job execution. The life is terminated either:

  - by *FreeJobHandle* message from the client, or
  - by any implementation defined way. For example, a Functional Unit may be implemented such that it retains only the last twenty job results, a new one always replacing the oldest one.

  The server is desirable to retain JobEntryID and status of all job entries persistent.

The Life parameter is included in the job-request-type command to explicitly specify the life of JobHandle, JobEntryID, and job status. If the Life parameter is omitted, the life of "Job" is assumed.

In any case, the life is terminated if the job is canceled by the client before or during job execution.

1.4.4.1.3.Job Status Notification

A job or job entry is in one of the following state:

- Queued : execution has not begun
- Started : execution is in progress
- Suspended : execution is temporarily suspended
- Completed : execution has successfully completed
- Error : execution has completed in error
- WaitingForScheduledTime : waiting for the scheduled execution start time
- Canceled : queued job or job entry is canceled
- Aborted : execution is aborted

Following figure shows the relationship among job status and how job status is transited to another status.

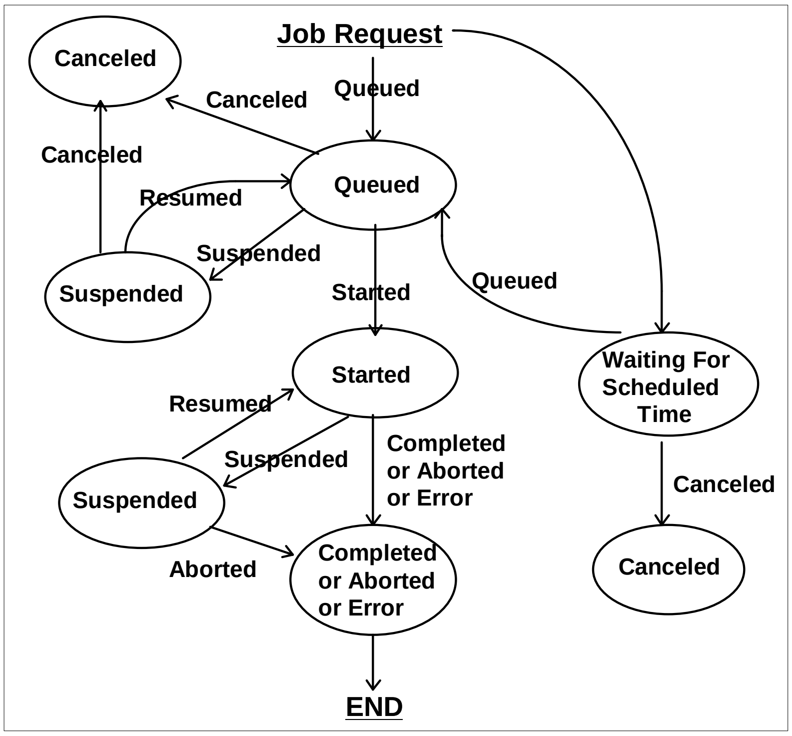


Fig 1 Job status transition table

Note

JobStatusCode includes both job status and the transaction. Only NotifyJobStatus uses the transaction part and the others, e.g. JobStatusNotificationMode/FUJobList, use the job status part. Above figure shows the job-status transaction table, a job status name in an oval and a transaction name along an arrow.

If the job consists of multiple job entries, how job entry status transition will affect to the job status will be described in each Functional Unit part.

The client can either solicit the server to know the current status of a job or job entries, or request the server to notify the client at certain status changes of a job or job entries.

When the client solicits the server for the status of a job, it sends a QueryJobStatus message to the server.

Client	**Server**
QueryJobStatus(JobHandle) =>	
	<= ACK(JobStatusCode, ...)

When the client solicits the server for the status of a specific job entry or all the job entries of a job, it sends a QueryJobEntryStatus message to the server.

Client	**Server**
QueryJobEntryStatus(JobHandle, JobEntryID) =>	
	<= ACK(JobStatusCode, ...)

When the client requests the server to notify certain status changes of a job or the job entries of a job:

- a **JobStatusNotificationMode** parameter is included in the job-request-type command, or
- the client sends StartMonitorJobStatus command which includes a JobStatusNotificationMode parameter. (The clients sends CancelMonitorJobStatus command when it no longer needs to be notified of job status changes.)

The JobStatusNotificationMode parameter specifies:

- what types of status changes should be notified, and
- whether the notification is requested for the job on the whole, or the notification is requested for each job entry of the job.

When the JobStatusNotificationMode indicates the notification is for the job on the whole:

Client	**Server**

COMMAND(..., JobStatusNotificationMode, ...) =>

<= ACK(JobHandle)

:

A notification is sent when the job execution is

- *started,*
- *suspended,*
- *resumed,*
- *completed successfully,*
- *completed in error,*
- *canceled, or*
- *aborted*

as requested:

<= NotifyJobStatus(JobHandle, JobStatusCode, ...)

ACK(NULL) =>

:

<= NotifyJobStatus(JobHandle, JobStatusCode, ...)

ACK(NULL) =>

:

When the JobStatusNotificationMode indicates the notification is for each job entry:

Client	**Server**

COMMAND(..., JobStatusNotificationMode, ...) =>

<= ACK(JobHandle)

:

A notification is sent when each job entry execution is

- *started,*
- *suspended,*
- *resumed,*
- *completed successfully,*
- *completed in error,*
- *canceled, or*
- *aborted*

as requested:

<= NotifyJobEntryStatus(JobHandle, JobEntryID, JobStatusCode, ...)

ACK(NULL) =>

<= NotifyJobEntryStatus(JobHandle, JobEntryID, JobStatusCode, ...)

ACK(NULL) =>

:

<= NotifyJobEntryStatus(JobHandle, JobEntryID, JobStatusCode, ...)

ACK(NULL) =>

<= NotifyJobEntryStatus(JobHandle, JobEntryID, JobStatusCode, ...)

ACK(NULL) =>

:

If the JobStatusNotificationMode parameter is not included in the job-request-type command, no notification is made unless the client issues a StartMonitorJobStatus command.

If the life of “job” or “persistent” is specified in the job-request-type command, the client application may close the service session after it receives the ACK for the command even if the job execution has not completed (or even begun). If any job status notifications are requested, the client application that is to receive the notifications must have registered itself with the Salutation Manager (SLM) as a [Client] Functional Unit.

By default, the job status notification is sent to the client application that submitted the job. Because the Functional Unit Handle of the job-requesting [Client] FU and the SLM-ID of the SLM with which the [Client] FU is registered are passed to the server FU at the *Open Service* request, the server FU will be able to send an *Open Service* request to the [Client] FU to send job status notifications if no service session exists.

It is possible for the job-requesting client application to specify that the job status notifications are sent to another [Client] FU. The notification-receiving application must have registered itself as a [Client] FU. The job-requesting client application specifies the Functional Unit Handle of the [Client] FU and the SLM-ID of the SLM with which the [Client] FU is registered in the job-requesting command as “NotificationScheme” parameter.

If the "check interval" parameter is specified in the job-request-type command, the Functional Unit has to request the Salutation Manager (SLM), by calling slmStartAvailabilityCheck(), to periodically check if the [Client] FU to receive the job status notifications is still available. If this parameter is not specified, no Availability Check is performed.

1.4.4.1.4.Job Control Attribute

Job-request-type commands sometimes have parameters called "**Job Control Attributes**". An example of a job control attribute is "Job Priority" attribute.

A particular job control attribute is associated with either the whole job or a job entry.

The following commands may be used to change the value of a job control attribute of a queued job:

- ChangeJobAttribute
- ChangeJobEntryAttribute

The specification of each command dictates which job control attributes support these commands.

1.4.4.1.5.Job Suspend/Resume

The following commands are defined to suspend the execution of queued or currently executing job or job entry:

- SuspendJob
- SuspendJobEntry

The following commands are defined to resume the suspended job or job entry:

- ResumeJob
- ResumeJobEntry

The specification of each job-request-type command dictates whether these commands are supported or not.

1.4.4.1.6.List FU Job Status

List FU Job Status type command is defined for the client to get the list of current job status in the Functional Unit. The supported command for each Functional Unit is defined in each chapter.

The example flow of the command and its response is as follows:

Client	Server
ListFUJob(...) =>	
	<= TransferDataBlock(FUJobList)
ACK(NULL) =>	

1.4.4.1.7.Skeleton of Job-Request-Type Command

Not all the parameters shown in the following skeleton of the job-request-type command are defined in each job-request-type command.

```
COMMAND                          ::= [APPLICATION tagCOMMAND] SEQUENCE
{
                                   COMPONENTS OF MsgHeader,
      : (other parameters)
   life                            [ ] Life                      DEFAULT job,
   jobStatusNotificationMode       [ ] JobStatusNotificationMode OPTIONAL,
                                       -- If omitted, no notification is made.
   notificationScheme              [ ] NotificationScheme        OPTIONAL,
                                       -- Omitted unless the job status notifications are to be
                                       -- sent to a [Client] FU other than the client that is
                                       -- sending this command
   checkInterval                   [ ] INTEGER                   OPTIONAL
                                       -- Interval (in seconds) for the FU-side SLM to periodically
                                       -- check the availability of the [Client] FU to receive the job
                                       -- status notification
                                       -- If omitted, the Availability Check is not performed.
   jobControlAttributes            [ ] ...
      : (other parameters)
}
```

1.4.4.2.Message Description

1.4.4.2.1.QueryJobStatus

The client sends this message to the server to query the status of a job.

It may be sent after the client receives a JobHandle in the response to a job-request-type command the client has issued, and before the life of JobHandle, JobEntryID, and job status is terminated.

Response

One of the following is returned in response to this message:

- *ACK*(JobStatusCode, ReasonCode)

  JobStatusCode indicates the status of the job. If JobStatusCode is either "error" or "suspended", ReasonCode is included and indicates the reason of the suspension or error.

- *NACK*(ReturnCode)

  The server has failed to process the message. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
QueryJobStatus                 ::= [APPLICATION tagQueryJobStatus] SEQUENCE
{
                                   COMPONENTS OF MsgHeader,
    jobHandle                      [0] JobHandle
}
```

1.4.4.2.2.QueryJobEntryStatus

The client sends this message to the server to query the status of a specific job entry or all the job entries of a job.

It may be sent after the client receives a JobHandle in the response to a job-request-type command the client has issued, and before the life of JobHandle, JobEntryID, and job status is terminated.

Response

One of the following is returned in response to this message:

- *ACK*(JobStatusCode, ReasonCode) or *ACK*(JobEntriesStatus)

  If the status of a specific job entry is queried, *ACK* includes JobStatusCode and optional ReasonCode. JobStatusCode indicates the status of the job entry. If JobStatusCode is either "error" or "suspended", ReasonCode is included and indicates the reason of the suspension or error.

  If the status of all the job entries is queried, *ACK* includes JobEntriesStatus.

- *NACK*(ReturnCode)

  The server has failed to process the message. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
QueryJobEntryStatus            ::= [APPLICATION tagQueryJobEntryStatus] SEQUENCE
{
                                   COMPONENTS OF MsgHeader,
    jobHandle                      [0] JobHandle,
    jobEntryId                     [1] JobEntryID    OPTIONAL
                                       -- If omitted, the status of all the job entries is requested.
}
```

```
JobEntriesStatus                ::= SET OF SEQUENCE
{
    jobEntryId                      [0] JobEntryID,
    jobEntryStatusCode              [1] JobStatusCode,
    jobEntryReasonCode              [2] ReasonCode              OPTIONAL
                                        -- present only if jobEntryStatusCode=suspended or error
}
```

1.4.4.2.3.NotifyJobStatus

The server sends *NotifyJobStatus* to the client to tell the job status change such as the completion of the job execution. The client has to indicate which types of job status changes it wishes to be notified by the JobStatusNotificationMode parameter of job-request-type command.

Response

One of the following is returned by the client in response to *NotifyJobStatus*:

- *ACK*(NULL)

  The client has received the notification successfully.

- *NACK*(ReturnCode)

  The client has failed to process the message. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
NotifyJobStatus                 ::= [APPLICATION tagNotifyJobStatus] SEQUENCE
{
                                    COMPONENTS OF MsgHeader,
    jobHandle                       [0] JobHandle,
    jobStatusCode                   [1] JobStatusCode,
    reasonCode                      [2] ReasonCode              OPTIONAL
                                        -- present only if jobStatusCode=suspended or error
}
```

1.4.4.2.4.NotifyJobEntryStatus

The server sends *NotifyJobEntryStatus* to the client to tell the job entry status change such as the completion of the job entry execution. The client has to indicate which types of job entry status changes it wishes to be notified by the job status notification mode parameter of job-request-type command.

Response

One of the following is returned by the client in response to *NotifyJobEntryStatus*:

- *ACK*(NULL)

  The client has received the notification successfully.

- *NACK*(ReturnCode)

  The client has failed to process the *NotifyJobStatus*. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
NotifyJobEntryStatus          ::= [APPLICATION tagNotifyJobEntryStatus] SEQUENCE
{
                                  COMPONENTS OF MsgHeader,
    jobHandle                     [0] JobHandle,
    jobEntryId                    [1] JobEntryID,
    jobStatusCode                 [2] JobStatusCode,
    reasonCode                    [3] ReasonCode          OPTIONAL
                                      -- present only if jobStatusCode=suspended or error
}
```

1.4.4.2.5.ChangeJobAttribute

The client sends this message to change the value of a job control attribute parameter of a queued job-request-type command.

If the execution of the specified job has already been completed, the command is rejected by NACK (ReturnCode = rcJobAlreadyExecuted) response. (ReturnCode = rcInvalidJobHandle is used if the life of JobHandle has already expired.)

If the specified job is being executed, it is implementation dependent whether:

- the command is rejected by NACK (ReturnCode = rcJobAlreadyExecuted) response, or
- the command is accepted and the job is executed under the updated attribute value.

Response

One of the following is returned by the server in response to this message:

- *ACK*(NULL)

  The server has updated the job control attribute value successfully.

- *NACK*(ReturnCode)

  The server has failed to process the message. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
ChangeJobAttribute            ::= [APPLICATION tagChangeJobAttribute] SEQUENCE
{
                                  COMPONENTS OF MsgHeader,
    jobHandle                     [0] JobHandle,
    attributeId                   [1] AttributeID,
    attributeValue                [2] ANY
}
```

1.4.4.2.6.ChangeJobEntryAttribute

The client sends this message to change the value of a job control attribute parameter of a job entry of a queued job-request-type command.

If the execution of the specified job entry has already been completed, the command is rejected by NACK (ReturnCode = rcJobAlreadyExecuted) response. (ReturnCode = rcInvalidJobHandle is used if the life of JobHandle has already expired.)

If the specified job entry is being executed or if the specified job is being executed but the execution of the specified job entry has not begun, it is implementation dependent whether:

- the command is rejected by NACK (ReturnCode = rcJobAlreadyExecuted) response, or
- the command is accepted and the job entry is executed under the updated attribute value.

Response

One of the following is returned by the server in response to this message:

- *ACK*(NULL)

  The server has updated the job control attribute value successfully.

- *NACK*(ReturnCode)

  The server has failed to process the message. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
ChangeJobEntryAttribute       ::= [APPLICATION tagChangeJobEntryAttribute] SEQUENCE
{
                                  COMPONENTS OF MsgHeader,
   jobHandle                      [0] JobHandle,
   jobEntryId                     [1] JobEntryID,
   attributeId                    [2] AttributeID,
   attributeValue                 [3] ANY
}
```

1.4.4.2.7.SuspendJob

The client sends this message to suspend the execution of a queued or currently executing job-request-type command.

Response

One of the following is returned by the server in response to this message:

- *ACK*(NULL)

  The server has suspended the job successfully.

- *NACK*(ReturnCode)

  The server has failed to process the message. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
SuspendJob                    ::= [APPLICATION tagSuspendJob] SEQUENCE
{
                                  COMPONENTS OF MsgHeader,
   jobHandle                      [0] JobHandle
}
```

1.4.4.2.8.SuspendJobEntry

The client sends this message to suspend the execution of a queued or currently executing job entry of a job-request-type command.

Response

One of the following is returned by the server in response to this message:

- *ACK*(NULL)

  The server has suspended the job entry successfully.

- *NACK*(ReturnCode)

  The server has failed to process the message. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
SuspendJobEntry               ::= [APPLICATION tagSuspendJobEntry] SEQUENCE
{
                                  COMPONENTS OF MsgHeader,
   jobHandle                      [0] JobHandle,
   jobEntryId                     [1] JobEntryID
}
```

1.4.4.2.9.ResumeJob

The client sends this message to resume a suspended job-request-type command.

Response

One of the following is returned by the server in response to this message:

- *ACK*(NULL)

  The server has processed the message successfully.

- *NACK*(ReturnCode)

  The server has failed to process the message. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
ResumeJob                     ::= [APPLICATION tagResumeJob] SEQUENCE
{
                                  COMPONENTS OF MsgHeader,
   jobHandle                      [0] JobHandle
}
```

1.4.4.2.10.ResumeJobEntry

The client sends this message to resume a suspended job entry of a job-request-type command.

Response

One of the following is returned by the server in response to this message:

- *ACK*(NULL)

  The server has processed the message successfully.

- *NACK*(ReturnCode)

  The server has failed to process the message. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
ResumeJobEntry              ::= [APPLICATION tagResumeJobEntry] SEQUENCE
{
                                COMPONENTS OF MsgHeader,
    jobHandle                   [0] JobHandle,
    jobEntryId                  [1] JobEntryID
}
```

1.4.4.2.11.CancelJob

The client sends this message to cancel a job-request-command it has issued before.

It may be sent after the client receives a JobHandle in the response to a job-request-type command the client has issued, and before the life of JobHandle is terminated.

This message includes "abort" flag. If the flag is set, the job is aborted even if the job is being executed. If the flag is NOT set and the server has already started the execution of the job, *CancelJob* message is rejected by NACK (ReturnCode = rcJobAlreadyExecuted) response.

Response

One of the following is returned by the server in response to this message:

- *ACK*(NULL)

  The server has canceled the job successfully.

- *NACK*(ReturnCode)

  The server has failed to process the message. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
CancelJob                     ::= [APPLICATION tagCancelJob] SEQUENCE
{
                                  COMPONENTS OF MsgHeader,
    jobHandle                     [0] JobHandle,
    abort                         [1] BOOLEAN
                                      -- if TRUE, job is canceled either queued or being executed
                                      -- if FALSE, job is canceled only if execution has not started
}
```

1.4.4.2.12.CancelJobEntry

The client sends this message to cancel a job entry in a job-request-command it has issued before.

It may be sent after the client receives a JobHandle in the response to a job-request-type command the client has issued, and before the life of JobHandle is terminated.

This message includes "abort" flag. If the flag is set, the job entry is aborted even if the job entry is being executed. If the flag is NOT set and the server has already started the execution of the job entry, *CancelJobEntry* message is rejected by NACK (ReturnCode = rcJobAlreadyExecuted) response. The support of this feature is optional. If the server implementation does not support it, *CancelJobEntry* message with abort=TRUE received while the job entry is being executed is rejected by NACK (ReturnCode = rcJobAlreadyExecuted).

Response

One of the following is returned by the server in response to this message:

- *ACK*(NULL)

  The server has canceled the job entry successfully.

- *NACK*(ReturnCode)

  The server has failed to process the message. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
CancelJobEntry                ::= [APPLICATION tagCancelJobEntry] SEQUENCE
{
                                  COMPONENTS OF MsgHeader,
    jobHandle                     [0] JobHandle,
    jobEntryId                    [1] JobEntryID,
    abort                         [2] BOOLEAN
                                      -- if TRUE, job is canceled either queued or being executed
                                      -- if FALSE, job is canceled only if execution has not started
}
```

1.4.4.2.13.FreeJobHandle

The client sends this message to tell the server that the JobHandle, JobEntryID, and job result no longer need to be retained.

It may be sent after the client receives a JobHandle in the response to a job-request-type command the client has issued, and before the life of JobHandle is terminated.

However, if the specified job execution has not been started or completed either successfully or unsuccessfully, the command is rejected by NACK (ReturnCode = rcJobNotCompleted).

In practice, this message is used by the client after the client first solicits the result of job execution by QueryJob(Entry)Status message when the life of job is "Persistent".

Response

One of the following is returned by the server in response to this message:

- *ACK*(NULL)

  The server has processed the command successfully.

- *NACK*(ReturnCode)

  The server has failed to process the message. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
FreeJobHandle                ::= [APPLICATION tagFreeJobHandle] SEQUENCE
{
                                 COMPONENTS OF MsgHeader,
    jobHandle                    [0] JobHandle
}
```

1.4.4.2.14.StartMonitorJobStatus

The client sends *StartMonitorJobStatus* to the server to be notified of the job status change such as the completion of the job execution. The client has to indicate which types of job status changes it wishes to be notified with JobStatusNotificationMode parameter. The StartMonitorJobStatus resets the FU’s jobStatusNotificationMode. For example, the StartMonitorJobStatus’s JobStatusNotificationMode overrides the job-request-type command’s jobStatusNotificationMode if any.

Response

One of the following is returned by the server in response to *StartMonitorJobStatus* :

- *ACK*(NULL)

  The server has received the command successfully.

- *NACK*(ReturnCode)

  The server has failed to process the message. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
StartMonitorJobStatus            ::= [APPLICATION tagStartMonitorJobStatus] SEQUENCE
{
                                     COMPONENTS OF MsgHeader,
    jobHandle                        [0] JobHandle,
    jobStatusNotificationMode        [1] JobStatusNotificationMode
}
```

1.4.4.2.15.CancelMonitorJobStatus

The client sends *CancelMonitorJobStatus* to the server to cancel the job-status-monitoring.

Response

One of the following is returned by the server in response to *CancelMonitorJobStatus* :

- *ACK*(NULL)

  The server has received the command successfully.

- *NACK*(ReturnCode)

  The server has failed to process the message. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
CancelMonitorJobStatus           ::= [APPLICATION tagCancelMonitorJobStatus] SEQUENCE
{
                                     COMPONENTS OF MsgHeader,
    jobHandle                        [0] JobHandle
}
```

1.4.4.2.16.List FU Job Status type command

The client sends a *List FU Job Status type* message to get the list of current job in the Functional Unit. The list is transferred from the Functional Unit to the client by using a Data Transfer message Sequence.

The list of job is transferred as ‘data’ consisting of one data block which may be split into multiple data block segments. Each data block segment is FUJobDescription (The format of each FUJobDescription is defined in each FU.) which is defined as SET OF these Descriptions. For example, if Functional Unit has 900 jobs, the Description of the first 300 jobs may be sent in the first data block segment, next 300 jobs in the second data block segment, and last 300 jobs in the last data block segment. Although each data block segment contains only a part of the whole folder set, the receiving application can decode each data block segment without waiting for the next data block segment.

If the specified parameter is unknown or not supported by the FU, the command is rejected by NACK.

Response

One of the following is returned in response to this message:

- *ACK*(ANY)

The client has successfully processed the received data.

- *NACK*(ReturnCode)

  The server has failed to process the command. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

Refer to each Functional Unit description.

1.4.5.Dynamic Status Messages

1.4.5.1.Overview

A **Dynamic Status Parameter** describes an aspect of the current Functional Unit status. There are three ways for the client to access dynamic status parameters:

- Query Dynamic Status

  The client may query the current value of a particular dynamic status parameter.

- Event Notification

  The client may request the Functional Unit to notify the client of any changes of the value of a particular dynamic status parameter.

- List Job Status

  The client may request the Functional Unit to get the list current job status.

The supported Dynamic Status ID is defined in the Capability Attribute.

1.4.5.1.1.Query Dynamic Status

QueryDynamicStatus command is defined for the client to get the current value of a particular Dynamic Status Parameter which describes an aspect of the Functional Unit status.

The flow of the command and its response is as follows:

Client	**Server**
when successful:	
QueryDynamicStatus(DynamicStatusID) =>	
	<= ACK(*value*)
when unsuccessful:	
QueryDynamicStatus(DynamicStatusID) =>	
	<= NACK(ReturnCode)

1.4.5.1.2.Event Notification

The client first subscribes to one or more dynamic status parameters by sending a SubscribeEvent command to the Functional Unit.

When the value of a subscribed dynamic status parameter changes, the Functional Unit notifies the client of the new value of the dynamic status parameter by sending a NotifyEvent message.

The client sends an UnsubscribeEvent command to the Functional Unit if it no longer wishes to be notified of the changes of the dynamic status parameter value(s).

The normal flow of messages is as follows:

Client	**Server**

first, client subscribes to dynamic status parameter(s):

SubscribeEvent(list of DynamicStatusIDs, ...) =>

<= ACK(SubscriptionHandle)

:

whenever the value of a subscribed dynamic status parameter changes:

<= NotifyEvent(SubscriptionHandle, DynamicStatusID, *value*)

ACK(NULL) =>

:

<= NotifyEvent(SubscriptionHandle, DynamicStatusID, *value*)

ACK(NULL) =>

:

when the client no longer wishes to be notified of changes:

UnsubscribeEvent(SubscriptionHandle) =>

<= ACK(NULL)

It may take a very long time before the client is notified of an event after it subscribes to the event. If the client intends to terminate the service session after it sends a SubscribeEvent command, the “life” parameter in the SubscribeEvent command must be set to “persistent”. While such a long-term subscription is effective, the availability of the client and the Functional Unit must be periodically checked using the Salutation Manager’s Availability Check function so that:

- the client can be assured that the Functional Unit has not forgotten (e.g. by crush or re-start) that the client is still waiting for the notification of the event, and
- the Functional Unit can be assured that the client is still up and running (e.g. not crushed) to receive a notification of the event.

If the “life” parameter in a SubscribeEvent command is set to “session”, the subscription is automatically canceled when the service session is terminated. In this case, the “checkInterval” parameter of the SubscribeEvent command must be omitted, and Availability Check is not performed.

Please refer to the relevant sections in the part-1 of the specification for the detail of Salutation Manager Protocol and the Salutation API definitions for the Availability Check. This section describes the flow of operations for the client and the Functional Unit to initiate and end the Availability Check.

When a client application intends to send a SubscribeEvent command with “life = persistent” parameter, it must have registered itself as a [Client] Functional Unit with the Salutation Manager (SLM). In the following description, the [Client] Functional Unit sending a SubscribeEvent is called "Client-FU", and the Functional Unit to which the SubscribeEvent command is sent is called "Server-FU".

1) Before the Client-FU sends the SubscribeEvent to the Server-FU, it calls **slmStartAvailabilityCheck()** with the following parameters:

    - AvailabilityCheckMode = **Receiver**
    - SLM-ID of the SLM with which the Server-FU is registered
    - FU Handle of the Server-FU
    - Check Interval
    - FU Handle of the Client-FU (i.e. its own FU Handle)
    - The entry point of the Client-FU's call-back function which will be called by the SLM when the Server-FU is found to be unavailable.

    slmStartAvailabilityCheck() returns a AvailabilityCheckHandle to the Client-FU. The Client-FU must retain the AvailabilityCheckHandle associated with this SubscribeEvent request, so that when it cancels the SubscribeEvent request later, it can also cancel this Availability Check request.

2) Then, the Client-FU sends the SubscribeEvent to the Server-FU including the following parameters:

    - Check Interval (the same value as that is specified in **slmStartAvailabilityCheck()**)

3) The Server-FU processes the received SubscribeEvent, and returns a response back to the Client-FU.

    If the response is negative (i.e. NACK), the Client-FU must call **slmCancelAvailabilityCheck()** with the AvailabilityCheckHandle which was given by the previous **slmStartAvailabilityCheck()**. Otherwise, proceed to the following.

    If the response is positive (i.e. ACK), the Server-FU calls **slmStartAvailabilityCheck()** with the following parameters:

    - AvailabilityCheckMode = **Sender**
    - SLM-ID of the SLM with which the Client-FU is registered
    - FU Handle of the Client-FU
    - Check Interval : copied from the SubscribeEvent
    - FU Handle of the Server-FU (i.e. its own FU Handle)
    - The entry point of the Server-FU's call-back function which will be called by the SLM when the Client-FU is found to be unavailable.

slmStartAvailabilityCheck() returns a AvailabilityCheckHandle to the Server-FU. The Server-FU must retain the AvailabilityCheckHandle associated with this SubscribeEvent request, so that when the SubscribeEvent request is canceled, it can also cancel this Availability Check request.

4) The Availability Check is performed between the client-side SLM and the server-side SLM (which can be the same SLM). As long as the Client-FU and the Server-FU are available, the call-back function of the Client-FU or the Server-FU is not called.

   If the Server-FU is found to be unavailable, the call-back function of the Client-FU is called. The Client-FU must call **slmCancelAvailabilityCheck()** with the SubscriptionHandle given by the previous **slmStartAvailabilityCheck()**.

   If the Client-FU is found to be unavailable, the call-back function of the Server-FU is called. The Server-FU must call **slmCancelAvailabilityCheck()** with the SubscriptionHandle given by the previous **slmStartAvailabilityCheck()**. The Server-FU invalidates the event subscription from the Client-FU.

5) When the Client-FU no longer needs to be notified of the event, it sends an UnsubscribeEvent command to the Server-FU, and calls **slmCancelAvailabilityCheck()** with the AvailabilityCheckHandle which was given by the previous **slmStartAvailabilityCheck()**.

6) When the Server-FU receives the UnsubscribeEvent command from the Client-FU, it calls **slmCancelAvailabilityCheck()** with the AvailabilityCheckHandle which was given by the previous **slmStartAvailabilityCheck()**.

1.4.5.2.Message Description

1.4.5.2.1.QueryDynamicStatus

The client sends a *QueryDynamicStatus* message to get the current value of a dynamic status parameter. The message contains a Dynamic Status ID as a parameter.

If the specified dynamic status parameter is unknown or not supported by the FU, the command is rejected by NACK.

Response

One of the following is returned in response to this message:

- *ACK*(ANY)

  The server has successfully processed the command and is returning the value of the specified dynamic status parameter.

- *NACK*(ReturnCode)

  The server has failed to process the command. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
QueryDynamicStatus              ::= [APPLICATION tagQueryDynamicStatus] SEQUENCE
{
                                    COMPONENTS OF MsgHeader,
   dynamicStatusId                  [0] DynamicStatusID
}
```

1.4.5.2.2.SubscribeEvent

The client sends a *SubscribeEvent* message to request the Functional Unit to notify the client of any changes of the value of particular dynamic status parameter(s).

If any of the specified dynamic status parameters are unknown or not supported by the FU, the entire command is rejected by NACK.

"Life" parameter indicates whether the subscription is valid only during the current service session (life = "session") or the subscription should be retained even after the current service session is terminated (life = "persistent"). “job” shall not be set to the life parameter.

If the life of “persistent” is specified in the command, the client application may close the service session after it receives the ACK for the command. The client application that is to receive the event notification must have registered itself with the SLM as a [Client] Functional Unit.

If “life = persistent”, the “checkInterval” parameter must be present. If “life = session”, the “checkInterval” parameter must be omitted.

The event notification is sent to the client application that submitted the *SubscribeEvent* command. Because the Functional Unit Handle of the [Client] FU and the SLM-ID of the SLM with which the [Client] FU is registered are passed to the server FU at the *Open Service* request, the server FU will be able to send an *Open Service* request to the [Client] FU to send the event notification if no service session exists.

If "life = persistent” parameter is specified in the *SubscribeEvent* command, the Functional Unit has to request the Salutation Manager (SLM), by calling slmStartAvailabilityCheck(), to periodically check if the [Client] FU to receive the event notification is still available.

Response

One of the following is returned in response to this message:

- *ACK*(SubscriptionHandle)

  The server has successfully processed the command and is returning a handle that uniquely identifies this particular subscription.

- *NACK*(ReturnCode)

  The server has failed to process the command. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
SubscribeEvent                  ::= [APPLICATION tagSubscribeEvent] SEQUENCE
{
                                    COMPONENTS OF MsgHeader,
    eventList                       [0] SET OF DynamicStatusID,
    life                            [1] Life,
    checkInterval                   [2] INTEGER              OPTIONAL
                                        -- Interval (in seconds) for the FU-side SLM to periodically
                                        -- check the availability of the [Client] FU to receive the job
                                        -- status notification
                                        -- This parameter shall be present if life = persistent.
                                        -- This parameter shall be omitted if life = session.
}
```

1.4.5.2.3.NotifyEvent

The Functional Unit sends a *NotifyEvent* message to the client to notify the new value of a dynamic status parameter that the client has previously subscribed to, when the value has changed.

If the indicated SubscriptionHandle or DynamicStatusID is unknown to the client, the message is rejected by NACK.

Response

One of the following is returned in response to this message:

- *ACK*(NULL)

  The client has successfully accepted the notification.

- *NACK*(ReturnCode)

  The client has failed to process the message. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
NotifyEvent                     ::= [APPLICATION tagNotifyEvent] SEQUENCE
{
                                    COMPONENTS OF MsgHeader,
    subscriptionHandle              [0] SubscriptionHandle,
    dynamicStatusId                 [1] DynamicStatusID,
    dynamicStatusValue              [2] ANY
}
```

1.4.5.2.4.UnsubscribeEvent

The client sends an *UnsubscribeEvent* message to the Functional Unit when it no longer wishes to be notified of any changes of the value of subscribed dynamic status parameter(s).

If the specified subscription handle is unknown to the FU, the command is rejected by NACK.

Response

One of the following is returned in response to this message:

- *ACK*(NULL)

The server has successfully processed the command and canceled the specified subscription.

- *NACK*(ReturnCode)

  The server has failed to process the command. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
UnsubscribeEvent          ::= [APPLICATION tagUnsubscribeEvent] SEQUENCE
{
                              COMPONENTS OF MsgHeader,
    subscriptionHandle        [0] SubscriptionHandle
}
```

1.4.6.Vendor Escape

1.4.6.1.Overview

There may be cases that vendor-unique commands and responses are necessary to implement vendor-unique functions in each Functional Unit. The VendorEscape message may be used to implement such functions. The Manufacturer, Product, and Version attributes which may be included in any Functional Unit Description Records help vendor-unique Salutation applications to determine if and what vendor-unique functions are supported.

1.4.6.2.Message Description

1.4.6.2.1.VendorEscape

The number and data type of parameters in the VendorEscape message and its response are to be defined by each vendor's specification, and out of the scope of this document.

VendorEscape is unique to vendors, so, the sender of VendorEscape command should check the validity of this command prior to issuing it. The sender can know it by examining the Manufacturer, Product and/or Version in the query capability.

Response

One of the following is returned in response to this message:

- *ACK*(...)

  The server has successfully processed the command.

- *NACK*(ReturnCode)

  The server has failed to process the command. *NACK* includes a **ReturnCode** which indicates the reason of the failure.

ASN.1 Syntax Definition

```
VendorEscape              ::= [APPLICATION tagVendorEscape] SEQUENCE
{
                              COMPONENTS OF MsgHeader,
    parameter                 [0] ANY
}
```

1.5.Common Attributes

The following table shows the attributes that are **mandatory** in all the **registered** Functional Unit Description Records regardless of the type of Functional Unit.

Capability Attribute Name	ID	Data Type	Compare Function ID
Major version	10	INTEGER	intEqualTo
Minor version	11	INTEGER	intGreaterThanOrEqualTo
Default coded character set	20	CharSetID	intEqualTo
FU name	30	DisplayString (SIZE(0..63))	strEqualTo
Manufacturer name	40	DisplayString (SIZE(0..63))	strEqualTo
Manufacturer product name	41	DisplayString (SIZE(0..63))	strEqualTo
Manufacturer product version	42	DisplayString (SIZE(0..63))	strEqualTo
Physical location	50	DisplayString (SIZE(0..255))	strEqualTo
Contact person name	51	DisplayString (SIZE(0..255))	strEqualTo
Authentication flavors	60	SET OF AuthenticationFlavor	setIntDoesContain

Note : Major/minor version attributes indicate the version number of the Salutation Personality Protocol architecture definition of the Functional Unit. When these two attributes are included in a requested FUDR, the requested FUDR matches the registered FUDR if the two FUDRs have the same major version number AND the minor version number of the registered FUDR is greater than or equal to that of the requested FUDR.

Under the initial version of the Salutation Architecture, the following values must be specified in any registered FUDRs:

- Major version = 2
- Minor version = 0

Additional attributes are defined for each type of Functional Unit. All the defined attributes must be present in a registered FUDR.

1.6.National Language Support

This section defines how character sets are encoded under the Salutation Architecture. Any other internationalization considerations are discussed under the specification of each Functional Unit as required.

The following textual information are subject to the definition in this section.

- **Attribute** values of textual data type in the Functional Unit Description Records
- **Parameter**s of textual data type in commands and responses under the Salutation Personality Protocol.
- **“Data”** of textual data type, e.g. plain text, unless the coded character set is identified by another method, e.g.
    - ☐ the architecture/specification of data format defines its own coded character set

identification rule, or

- ☐ coded character set identification rule is defined by the specification of Functional Unit

For example, versit's **vCard** data format has its own coded character set identifier, therefore, this is outside of the scope of this section's definition. Refer to the Versit documents listed in the "References" section on page 275 for the detail of versit's vCard data format.

These are called target attributes, parameters and "data" in the following description.

Note: The architecture only defines the coded character set identification rule for interchange. Implementations may choose whichever coded character set they like for internal usage.

DisplayString data type is defined by the architecture as follows. It should be used for any textual data definition for the target attributes and parameters.

```
DisplayString                ::= OCTET STRING        -- Textual information
```

CharSetID data type is defined by the architecture as follows to identify a particular coded character set.

```
CharSetID                    ::= INTEGER         -- Coded character set ID as registered with IANA
                                                 -- (MIB enumeration value is used)
```

The value of CharSetID is an integer registered with the Internet Assigned Numbers Authority (IANA) as MIB enumeration value to identify a coded character set. The IANA's registry of the coded character sets can be found at:

ftp://venera.isi.edu/in-notes/iana/assignments/character-sets

The architecture recommends implementations to choose coded character sets for interchange from the following list:

Coded Character Set	Countries	IANA-registered character set name	Value of CharSetID
ISO 8859-1	Latin-1 countries	ISO_8859-1:1987	4
ISO 10646-1 level 3 (Unicode)	Countries using double-byte characters	ISO-10646-UCS-2	1000
JIS X0208-1978 (Shift JIS)	Japan	Shift_JIS	17

Note: This list may grow based on requirements.

The following attribute is defined for the Functional Unit Description Record (FUDR):

- Default Coded Character Set

Data Type : CharSetID

All the target attributes of DisplayString data type in the FUDR are encoded in the coded character set specified by this attribute.

1.7.[Client] Functional Unit

1.7.1.Overview

In general, Functional Units are abstraction of server applications and provide services to client applications or other Functional Units. However, a special Functional Unit, called **[Client] Functional Unit**, is defined to represent certain client applications rather than server applications.

The [Client] Functional Unit is semantically different from other Functional Units as it represents a client, not server, application. However, the Salutation Manager tre