

HP JetSend™ Communications Technology

Appliance Development Kit Development Guide

HP product # C3093A

HEWLETT PACKARD

Version 1.0

JetSend™ is a registered trademark of Hewlett-Packard Company.

For information about HP JetSend and for periodic updates for the appliance development kit, see our web site at **http://www.jetsend.hp.com**.

This document is supplied on an “as is” basis with no warranty and no support. HP makes no express warranty, whether written or oral with respect to this specification. HP DISCLAIMS ALL IMPLIED WARRANTIES INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

LIMITATION OF LIABILITY: IN NO EVENT SHALL HP BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFIT OR LOST DATA) WHETHER BASED ON WARRANTY, CONTRACT, TORT OR ANY OTHER LEGAL THEORY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL.

This document contains information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another language without the prior written consent of Hewlett-Packard Company.

The information contained in this document is subject to change without notice.

No trademark, copyright, or patent licenses are expressly or implicitly granted (herein) with this specification.

© Copyright 1997, Hewlett-Packard Company

Trademarks: All brand names and product names used in this ADK are trademarks, registered trademarks, or trade names of their respective holders. Hewlett-Packard is not associated with any product or vendor mentioned in this ADK.

Table of Contents

Appliance Development Kit—Quick Start 1-1
Introduction. 1-1
Road Map. 1-1
Start Up 1-1
Architectural Design of Appliance 1-1
Determine Transport Mechanism 1-1
Determine E-Material Design. 1-2
Incorporate JetSend into Target System 1-2
Test 1-2
Certification 1-2
Description 1-2
JetSend ADK Quick Reference 1-6
ADK CD Directory Structure 1-6
ADK File Reference 1-7
JetSend Core Files 1-7
JetSend E-Material Files 1-9
JetSend Support Files 1-10
Transport Layer Files. 1-11
Device-Code Sample Files 1-13
Other Files 1-13
JetSend Code Size. 1-14
HP JetSend Core Integration into an Appliance 1-16
JetSend-Enabled Appliance Requirements 2-1
Appliance 2-1
Required 2-1
Recommended 2-2
Development Hardware 2-3
Required 2-3
Recommended 2-3
Development Software. 2-3
Required 2-3
Recommended 2-3
JetSend-Enabled
Device Architecture. 3-1

Model of JetSend-Enabled Appliance Architecture 3-1
Components of a JetSend-Enabled Appliance. 3-2
Negotiating Layers. 3-2
Negotiation Services 3-3
Protocols 3-3
Application Programmatic Interfaces (API) 3-5
Device Specific API 3-5
E-Material Library API 3-5
Device Independent API 3-6
Transport Independent API 3-6
Transport Specific API. 3-6
Types of JetSend-Enabled Appliances 3-7
Single-Function Appliance. 3-7
Hybrid Appliance 3-7
Soft Appliance 3-8
Multifunction Appliance 3-8
JetSend Communications Model/Requirements. 4-1
Communication of JetSend-Enabled Appliances 4-1
Basic Surface Operations 4-1
Network Exchange View of E-Material 4-5
Buffer Management of Data. 4-6
Operation Expectations 4-8
Job Processing 4-9
Negotiation of Surface Information 5-1
Form of Appliance Communication 5-1
Surface Representation. 5-1
Surface Structure. 5-1
Well-Known Surfaces 5-2
Surface Impression and Surface Change 5-5
Negotiation between JetSend-Enabled Appliances. 5-6
E-Material Fundamentals 6-1
Surface Representation 6-1
E-Material Exchange 6-2
E-Material Characteristics 6-3
E-Material Surface Structure 6-4
Structure Within a Single Surface 6-4
Structure Linking Surfaces. 6-7
E-Material Surface. 6-9

Tabular View of E-Material Surface Descriptions 6-13
Code-Level View of E-Material 6-17
E-Material Type 6-18
Surface Hierarchy Examples 6-21
Detailed Document Example 6-24
JetSend Coding Fundamentals. 7-1
Datatype EMtype 7-1
Datatype EMvalue 7-2
Example - Creating a Description 7-2
Example - Memory and Description. 7-3
Example - Reading a Description 7-5
JetSend APIs 7-6
Transport Independent API (TIAPI) 7-6
System API 7-6
Device API 7-6
Device Independent API 7-6
Transport Independent API Reference Implementation 8-1
HP JetSend Sample Code 9-1
Overview 9-1
Notes on Sender Sample Programs 9-5
Notes on Receiver Sample Programs 9-10
Sample One Sender—InlineTextSender 9-17
Sample One Receiver - Inline Text 9-18
RcvText1 9-18
RcvTriggers1 9-18
Sample Two Sender—TextSender. 9-19
Sample Two Receiver—An Improved Text Receiver 9-20
Changes to the Common Files 9-20
Encoding Module 9-20
Sample Three Sender—FileSender 9-21
Sample Three Receiver—Receiving Files 9-22
Changes to the Common Files 9-22
Encoding Module 9-22
Sample Four Sender—ImageSender 9-23
Test files 9-23
Program Flow 9-24
Sample Four Receiver—Receiving Images 9-25
Changes to the Common Files 9-25

Encoding Module 9-25
Sample Five Sender—PlaneSender 9-26
Sample Five Receiver—Receiving Planes 9-27
Changes to the Common Files 9-27
Encoding Module 9-27
Sample Six Sender—AssocSender 9-29
Sample Six Receiver—Receiving Associations 9-30
Changes to the Common Files 9-30
Encoding Module 9-30
Testing Your JetSend Appliance 10-1
Introduction 10-1
A Three-Stage Approach 10-1
Testing JetSend Core in the Target Environment 10-1
Verification of JetSend Appliance Negotiation 10-41
Verification of JetSend Appliance Operation 10-44

JetSend E-Material Specification A-1
The Structure of E-Material A-1
E-Material Block A-1
Surface Description Block A-3
Document Description A-4
E-Material Fundamentals A-6
Data Types A-7
Representational Types A-8
Selection List, Decision Path, Encoding Hierarchy A-9
Child List, Reference List, Surface Hierarchy A-11
Required Attributes, Default Encodings A-13
Encoding for vImage Surface A-14
Definition for vImage A-14
Surface Description for vImage A-14
Description Request Block for vImage A-18
Description Reply Block for vImage A-19
Content Request Block for vImage A-21
Content Reply Block for vImage A-23
In-Line Content Data for vImage A-24
Guidelines for the E-Material Encoding for vImage A-24
Hierarchy Diagram for vImage A-25
Keywords for Attributes and Values Defined for vImage A-26

Image Compression Standards A-28
vImage Examples A-30
Encoding for vText A-34
Definition for vText. A-34
Surface Description for vText A-34
Description Request Block for vText A-36
Description Reply Block for vText A-36
Content Request for vText. A-37
Content Reply Block for vText A-38
In-Line Content Data for vText. A-39
Guidelines for the E-Material Encoding for vText A-39
Hierarchy Diagram for vText A-41
Keywords for Attributes and Values Defined for vText A-41
vText Examples A-43
Encoding for vFile. A-45
Definition for vFile A-45
Surface Description for vFile A-45
Description Request Block for vFile. A-47
Description Reply Block for vFile A-48
Content Request for vFile A-49
Content Reply Block for vFile A-50
Guidelines for the E-Material Encoding for vFile A-51
Hierarchy Diagram for vFile A-51
Keywords for Attributes and Values Defined for vFile A-51
vFile Examples A-53
Encoding for vPlane A-54
Definition for vPlane. A-54
Surface Description for vPlane A-54
Description Request Block for vPlane A-60
Description Reply Block for vPlane A-61
Content Request for vPlane. A-63
Content Reply for vPlane. A-66
Hierarchy Diagram for vPlane A-70
Keywords for Attributes and Values Defined for vPlane A-70
Examples for vPlane A-72
Encoding for vAssociation A-76
Definition for vAssociation A-76
Surface Description for vAssociation A-76

Description Request Block for vAssociation A-78
Description Reply Block for vAssociation A-78
Content Request for vAssociation A-80
Content Reply for vAssociation A-83
Guidelines for an E-Material Encoding for vAssociation A-83
Hierarchy Diagram for the vAssociation Level A-85
vAssociation Examples A-86
General Guidelines for E-Material Encodings A-87

E-Material Library API Reference B-1
Activity Manager Reference C-1

Connection Activity Reference D-1

Push Send/Push Receiver Activity Reference E-1

Status Send/Status Receive Activity F-1

Address Send/Address Receive Activity G-1

Buffer Function Reference H-1

Queue Function Reference I-1

Miscellaneous Function Reference J-1

Transport Independent API Reference K-1

JetSend Multi-Network Address Specification 41
Syntax 41
Examples 42
Mulit-Port Network-Parallel Gateway Addressing 42

Glossary Glossary-1

Index Index-1

Chapter 1

Appliance Development Kit—Quick Start

Introduction

This document and the accompanying software have been created to help information appliance providers to incorporate the JetSend communications technology into an appliance. It is written for first-time JetSend developers.

This appliance development kit (ADK) also contains ANSI C code, which implements the JetSend protocol. This functionality is found in the JetSend core code; the code may be included in appliances, as well as the reference implementations of the TIAPI and RMTP.

Please refer to the following web site for any additional information:

http://www.jetsend.hp.com

Road Map

Start Up

- Become familiar with the JetSend concepts and architecture. (See Chapters 3 through 5.)
- Compile and run the JetSend samples along with the JetSend core and other components in a convenient development environment. (See Chapter 9.)

Architectural Design of Appliance

- Determine appliance design.
- Determine activities to be included and/or excluded.
- Most appliances will include the contact activity. Sending appliances will usually include the push-send, status-receive, and address activities. Receiving appliances will usually include the push-receive and status-send activities.

Determine Transport Mechanism

- The transport independent API is expected

- A reference implementation of the TIAPI to Berkeley sockets layer including RMTP is available in the ADK. These are to be used if IP is the underlying transport mechanism. UDP is a required part of Berkeley sockets. TCP is also used, if available. (See Chapter 8.)
- An appliance provider implementation of the TIAPI is required for other underlying transport mechanisms.

Determine E-Material Design

- The current ADK only supports static images.
- The sample code illustates the construction and imterpretation of the various encodings available for static images. This includes simple strings, planes, associations, and files, as well as images. An appliance provider is to implement device-specific code to manipulate and transform the content-specific data. The e-material libraries are available in the ADK. (See Chapter 6.)
- If an appliance design requires other types of e-material, this should be done in close cooperation with Hewlett-Packard JetSend support to insure peer support in devices.

Incorporate JetSend into Target System

- Port ANSI C into target system. The JetSend implementation offered in the ADK requires minimal complexity of the underlying operating system. In most ANSI C systems, code should compile directly.
- Header file review
- TIAPI implementation
- Device-specific implementation

Test

- Test the JetSend core in the target system
- Test the JetSend appliance against other appliances (See Chapter 10.)

Certification

- Certification tests are available as part of the ADK license. Refer to the JetSend web site for specific information on certification and the use of the JetSend logo.

Description

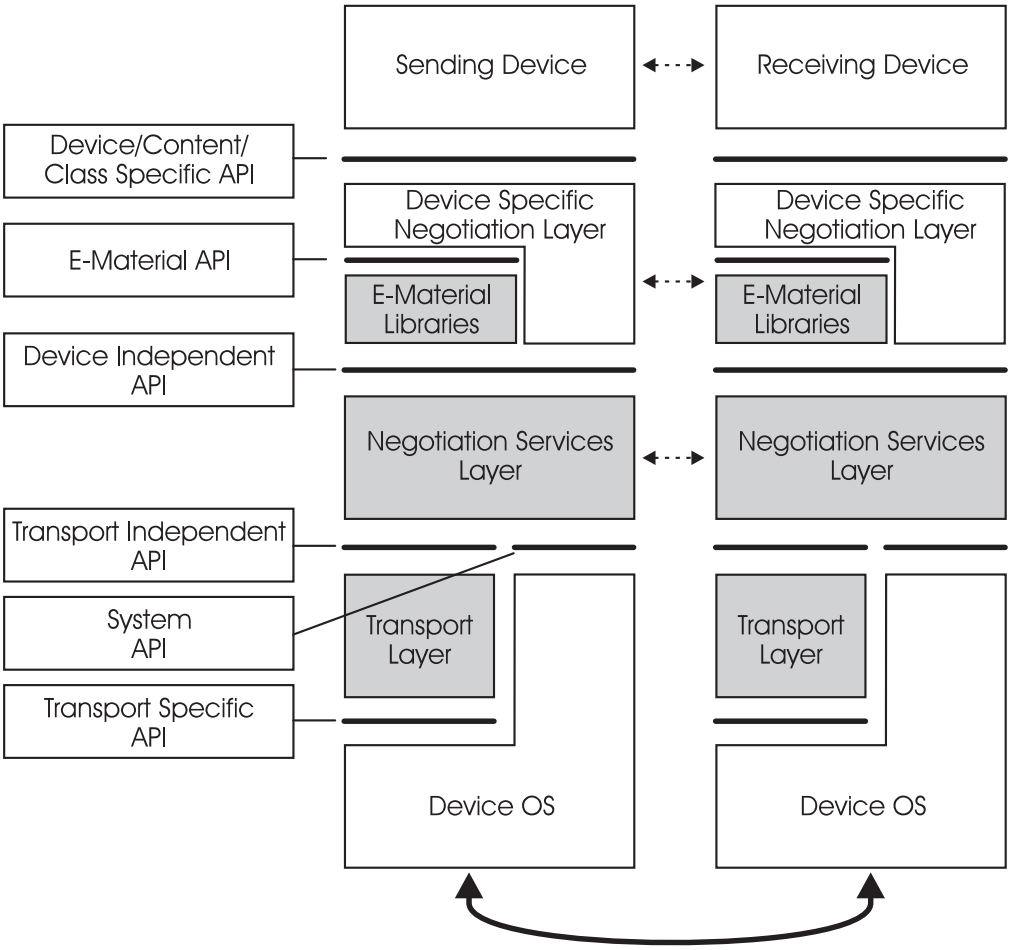
The *HP JetSend Communications Technology Appliance Development Kit* (ADK) helps **static image** appliance developers incorporate HP JetSend into their appliances. These appliances can be either hardware appliances, such as printers, scanners, and cameras, or soft appliances that run on a PC or other workstation.

HP JetSend utilizes **protocol stacks** to exchange information surfaces encoded as electronic material (e-material). The protocol allows appliances to **negotiate** the form of the e-material. This ADK guides the developer in joining the layers of the protocol stack with the existing native device. These **protocol layers** are inserted between existing information handling routines such as image capture, manipulation, or rendering, and existing network I/O, such as UDP/IP.

Figure 1.1 shows a simplified JetSend protocol stack. The shaded areas show which code is included within the ADK. The negotiation services layers form the JetSend core of the JetSend product. This code is fully tested and supported, and is available for Windows, HP-UX, Sun OS, and VxWorks. The thin transport layer includes a reference implementation that matches the **transport-independent API (TIAPI)** with Berkeley sockets; it also includes a reference implementation of **reliable message transport protocol (RMTP)**. RMTP gives an unreliable mechanism such as UDP/IP reliability without robbing it of its other connectionless characteristics. Both reference implementations can be used in appliances; other transport mechanisms must provide TIAPI and must be provided by the appliance developer.

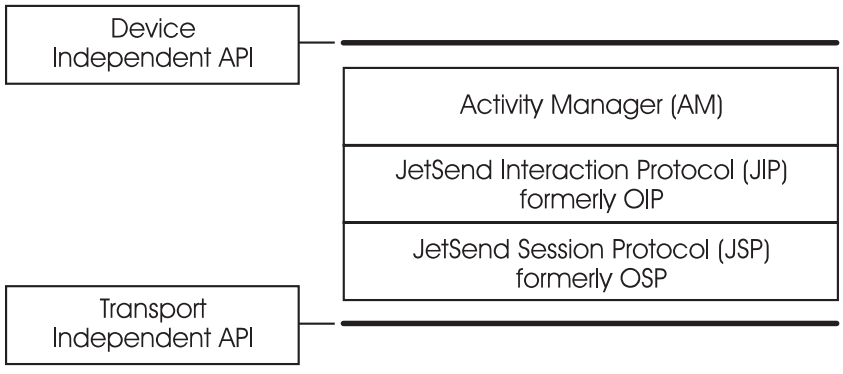
The sending device, receiving device, and device OS are assumed to be pre-existing peripheral and networking capabilities implemented in DOS, VxWorks, or some other ANSI C based system. The device-specific negotiation layer is supplied by the appliance developer, and marries the existing peripheral functionality to the JetSend stack. The e-material libraries and e-material specification are provided to aid in the creation and interpretation of e-material within this layer.

Figure 1.1 Simplified JetSend Protocol Stack



The underlying operating system is used by the negotiating services layers, also called the JetSend core or TOPS, to provide timer and memory allocation services. The negotiation services layers can generally be treated as a single unit. However, they are implemented as three separate layers in the JetSend core. The code modules that provide these layers are named in Figure 1.2.

Figure 1.2 JetSend Core Fine Structure

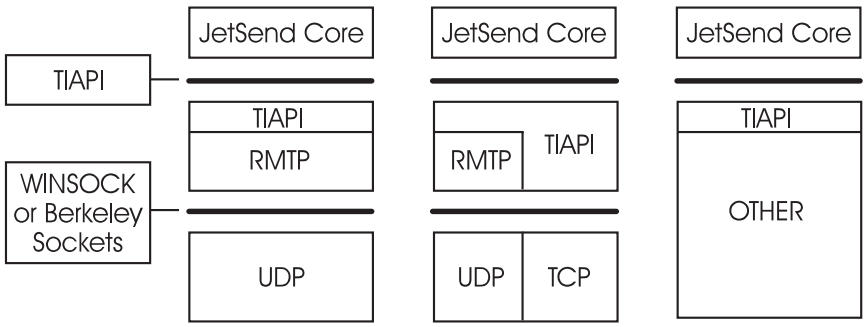


Note: During the early phases of its development, HP JetSend was called OPUS. The source files draw their names from The OPUS Protocol Stack (TOPS), OPUS Session Protocol

(OSP), and OPUS Interaction Protocol (OIP). JetSend is now the appropriate term for all situations, including JetSend core, JetSend Session Protocol (JSP), and JetSend Interaction Protocol (JIP). Vestiges of the OPUS name will still appear, particularly in code or in file names.

The transport layer implementation is highly dependent of the underlying transport. TIAPI is a simple API that can be implemented on virtually every asynchronous bi-directional transport.

Figure 1.3 JetSend Transport Alternatives



If the connection is through IP, the TIAPI must include the RMTP module, which communicates through UDP/IP. this utilizes datagrams as a simple, lightweight protocol. (See Figure 1.3.) This particular combination gives:

- reliability through RMTP
- connection through mechanisms with JSP (within the JetSend core)
- broadcast capability through UDP
- relatively lightweight implementation

If both TCP and UDP are available for an IP connection, the transport layer will take advantage of both. (See Figure 1.3.) The streams will be used for high volume transmission of content.

Other transports are not addressed in this ADK. However, the TIAPI is easily built on other bi-directional transports. (See Appendix K.)

JetSend ADK Quick Reference

ADK CD Directory Structure

./dev	Directory containing all ADK code
./dev/common	Contains JetSend code common to all devices/platforms
./dev/common/inc	JetSend header-files
./dev/common/src	JetSend source-files
./dev/contrib	Contains contributed sample code which uses the JetSend protocol.
./dev/contrib/misc	Miscellaneous sample code
./dev/contrib/tests	Source for the sample tests discussed in the testing section of the ADK
./dev/contrib/xchat	Source for a simple X Window/Motif JetSend desktop application
./dev/hpux	Build environment for compiling a JetSend device under HP-UX
./dev/hpux/exe	Destination for compiled executable
./dev/hpux/inc	Contains the header-files for an HP-UX specific transport layer as well as the “appliance” (appl*) code headers adjusted for the HP-UX environment
./dev/hpux/obj	Destination for compiled object cod.
./dev/hpux/src	Contains source code for an HP-UX specific transport layer as well as the “appliance” (appl*) source adjusted for the HP-UX environment
./dev/hpux/work	Location to store modified or “working” files
./dev/samples	Contains ADK sample code that makes use of the JetSend protocol to create simple sender/receiver pairs
./dev/utils	Contains various HP-UX scripts and utilities such as dos2ux, etc.
./dev/windows	Contains Microsoft Windows-specific code used for creating a JetSend.dll as well as code for building a transport layer for use in Windows
./dev/windows/dll	Files needed to build the JetSend dll
./dev/windows/exe	Destination for ADK sample executables
./dev/windows/inc	Windows-specific ADK sample header files. Includes reference transport layer headers
./dev/windows/libs	Destination for JetSend lib
./dev/windows/src	Windows-specific ADK sample source code. Includes reference transport layer code.
./doc	Directory containing all ADK documentation.

ADK File Reference

JetSend Core Files

The following JetSend files should not be modified. They constitute the "JetSend core" of the JetSend protocol and have been written to be as device independent as possible. This was done by using macros for system-specific calls—such as **malloc** and **free**—and for variable type definitions.

JetSend Core Headers

tops.h	JetSend core error-code defines, internal header-size macros, etc
tops_am.h	Activity Manager internal data-structures, macros, etc
tops_am_job.h	Activity Manager internal job-specific structures, macros
tops_am_list.h	Activity Manager internal queuing structures, prototypes
tops_device.h	The external Device API: prototypes, event-structures, etc
tops_oip.h	The internal JIP API: prototypes, macros, etc
tops_oiplib.h	JIP internal prototypes, structures, etc
tops_oipmsg.h	JIP internal message-structures, prototypes, structures, etc
tops_osp.h	The internal JSP API: prototypes, macros, structures, etc
tops_osp_protocol.h	JSP internal protocol structures, macros, prototypes, etc
tops_osppriv.h	JSP internal prototypes, structures, etc
ops_test.h	Needed for -DDEBUG JetSend compilation

JetSend Core Source

am.c	Activity Manager session-related (am*) API functions
am_ar.c	Activity Manager Address-Receiver (ar*) API functions
am_as.c	Activity Manager Address-Sender (as*) API functions
am_con.c	Activity Manager Contact (con*) API functions
am_list.c	Activity Manager internal list functions
am_oip.c	Activity Manager internal JIP function-callbacks
am_osp.c	Activity Manager addressing/gateway related functions
am_pr.c	Activity Manager Push-Receiver (pr*) API functions
am_ps.c	Activity Manager Push-Sender (ps*) API functions
am_sr.c	Activity Manager Status-Receiver (sr*) API functions

am_ss.c	Activity Manager Status-Sender (ss*) API functions
oip.c	JIP API functions
oiplib.c	JIP internal support functions
oipmsg.c	JIP internal message-related functions
osp.c	JSP API functions
osp_events.c	JSP internal event-handling functions
osp_protocols.c	JSP internal protocol functions
osp_support.c	JSP internal support functions
tops_test.c	JetSend test support functions (when -DDEBUG defined)

JetSend E-Material Files

The following files are used by the JetSend core code, but technically are not considered part of the JetSend protocol stack. These modules contain code for performing such duties as e-material creation.

Support Headers

appl_em.h	Internal generalized e-material function prototypes
appl_emlib.h	External e-material library API

Support Source

appl_em.c	JetSend internal generalized e-material functions
appl_emlib.c	External e-material API functions

JetSend Support Files

The following files are used by the JetSend core code, but technically are not considered part of the JetSend protocol stack. These modules contain code for performing such duties as list/queue management and buffer management. Some of these files may require modifications in order to map the JetSend function-macros and typedefs to a specific environment's corresponding values. For instance, appl.h defines what a "Tu16" (unsigned short) variable-type is in a specific environment. Refer to "HP JetSend Core Integration into an Appliance" on page 1-16 of the ADK documentation for more information.

Support Headers

appl.h	Contains device-dependent macro-definitions/typedefs, etc
appl_buf.h	JetSend buffer-management library header
appl_log.h	JetSend logging prototypes, defines, etc
appl_net.h	JetSend network-addressing function prototypes
appl_q.h	Internal JetSend queue-module types, prototypes, etc
appl_queue.h	External JetSend queue-module prototypes, etc
appl_rmtp.h	External RMTP API
appl_sys.h	JetSend system-specific macro translation prototypes
appl_tiapi.h	External TIAPI
appl_timer.h	JetSend system-specific timer function prototypes

Support Source

appl_buf.c	JetSend buffer-management API functions
appl_log.c	JetSend logging functions
appl_net.c	JetSend network-addressing/resolving functions
appl_q.c	Internal JetSend queue-module functions
appl_queue.c	External JetSend queue-module API functions
appl_sys.c	JetSend system-specific macro-translation functions
appl_timer.c	JetSend system-specific timer functions

Transport Layer Files

The files listed below are not part of the JetSend protocol stack. They provide a reference implementation of the transport layer required by the JetSend protocol. The JSP layer of the JetSend protocol stack uses the functions defined in the appl_tiapi.h file to listen, connect, send, and receive data across a transport. The RMTP modules provide the required support for a "reliable" message transport across UDP. The TIAPI modules provide a reference implementation for transport across an IP network.

Note: The following code is one way in which these functions can be coded. While they have been used in the HP-UX and Windows95/NT environment successfully, they are simply a reference implementation.

Transport Headers

rmtp_ack_pub.h	RMTP internal - prototypes for ack handling functions
rmtp_addrinfo_pub.h	RMTP internal - protypes for address info structures handling functions
rmtp_conn_pub.h	RMTP internal - prototypes for connection block handling
rmtp_connid_pub.h	RMTP internal - prototypes for connection ID alloc/dealloc functions
rmtp_dmninfo_pub.h	RMTP internal - prototypes for domain info alloc/dealloc functions
rmtp_module.h	Semi-public configuration parameters and structures
rmtp_queues_prv.h	RMTP internal - queue internal prototypes
rmtp_queues_pub.h	RMTP internal - queue external prototypes
rmtp_socketio_pub.h	RMTP internal - prototypes for socket I/O functions
rmtp_statemachine_pub.h	RMTP internal - prototypes for statemachine
rmtp_utils_pub.h	RMTP internal - prototypes for utilities
tiapi_cacheq_pub.h	TIAPI internal - prototype for allocation chaning (see tiapi_cacheq.c)
tiapi_module.h	Semi-public TIAPI configuration parameters and sturctures
tiapi_rply_pub.h	TIAPI internal - prototypes for reply building functions
tiapi_sockets.h	TIAPI definitions for interfacing to BSD sockets and TLI
tiapi_sockets_pub.h	TIAPI internal - prototypes for socket utilities
tiapi_stddefs.h	TIAPI standard macros - many small obscure syntatic-sugar macros
tiapi_utils_pub.h	TIAPI internal - prototypes for tiapi utils
tops_trplyQueue.h	TIAPI reference implementation header
tiapiprv.h	TIAPI reference implementation header
tops_unixtl.h	TIAPI reference implementation header

Transport Source

rmtp_ack.c	RMTP ack handling functions both send and receive
rmtp_addrinfo.c	RMTP addinto functions that store network addresses
rmtp_conn.c	RMTPTconnection blocks that contain connection state
rmtp_connid.c	RMTP alloc/free connection IDs (numbers)
rmtp_dmninfo.c	RMTP alloc/free domain info blocks that record info about network interfaces
rmtp_primary.c	RMTP all functions defined in RMTP API
rmtp_queues.c	RMTP functions that traverse queues used in RMTP
rmtp_socketio.c	RMTP low-level send and receive for BSD Sockets and TLI
rmtp_statemachine.c	RMTP state transition machine
rmtp_utils.c	RMTP computing utilities: window size, roundtrip averaging, checksum
tiapi_cacheq.c	TIAPI functions to queue memory allocations instead of freeing - turned OFF
tiapi_primary.c	All functions defined in TIAPI
tiapi_rply.c	TIAPI reply management functions
tiapi_sockets.c	TIAPI utilities for working with network (port translation, address comparison, etc.
tiapi_utils.c	TIAPI utilities for setting up replies, stuffing words, bytes
trplyQueue.c	Handles queue function in TIAPI
tiapi.c	Streams portion of TIAPI

Device-Code Sample Files

Various sample files provided with the JetSend ADK can help illustrate how the JetSend protocol can be used. These files are single-threaded processes written in ANSI C, and can be compiled under both Windows and UNIX environments. Make files are included so that they can be built directly in various operating systems. Porting to other C-based operating systems should be direct and unambiguous. Refer to the sample code documentation found in Chapter 9 of the ADK development guide for more information.

Other Files

In addition to the sample files mentioned above, there have also been included other source files that use the JetSend protocol. These files have been included as-is, and are not to be considered part of theADK code. They are provided simply as another resource that may help implement the JetSend protocol in a specific device.

JetSend Code Size

Given below are the object-code sizes of the various JetSend modules. These modules were compiled under HP-UX running on an HP Model 735/125 Workstation. The compiles were done with the following cc-command:

```
cc -c -Aa -O -D_HPUX_SOURCE -DPLATFORM_Unix32
```

The .o files were run through the “size” utility. These sizes are only estimates.

JetSend Core Modules

am.o	3708 bytes
am_ar.o	2412 bytes
am_as.o	2613 bytes
am_con.o	5966 bytes
am_list.o	448 bytes
am_oip.o	1480 bytes
am_osp.o	868 bytes
am_pr.o	3120 bytes
am_ps.o	3816 bytes
am_sr.o	4400 bytes
am_ss.o	4432 bytes
oip.o	8508 bytes
oiplib.o	2360 bytes
oipmsg.o	6160 bytes
osp.o	12968 bytes
osp_events.o	11884 bytes
osp_protocols.o	11832 bytes
osp_support.o	14012 bytes

Total JetSend core-code size: 100987 bytes (96.6K)

JetSend Support Modules

appl_buf.o	660 bytes
appl_em.o	5740 bytes
appl_emlib.o	7804 bytes
appl_net.o	504 bytes
appl_q.o	1028 bytes
appl_queue.o	1204 bytes
appl_sys.o	540 bytes
appl_timer.o	1436 bytes

Total JetSend support-code size: 18916 bytes (18.5 K)

HP JetSend Core Integration into an Appliance

The following three main header files will be addressed: appl.h, tops.h, and tops_osp.h. These header files should be reviewed and possibly modified by anyone attempting to integrate the JetSend core code into their device. It is a basic assumption that appliance providers know what platform the system will be run on and what capabilities the device requires from JetSend core.

appl.h

The header file, appl.h, provides appliance providers with the ability to select a platform/sub-platform type via a #define such that when the code is compiled, OS-specific dependencies are taken into account. The platform/sub-platform #define should be included in the makefile as well.

The first section of appl.h contains system functions that the device may need to tailor. These have to do with stdlib functions that are aliased and used throughout JetSend core (such as malloc and free). The appliance provider should map the aliases to functions that are desirable if the pre-coded alias is not acceptable (see the PLATFORM_WIN95 defines for examples). The aliases requiring functions are:

Tmalloc	memory allocation. Requires the same parameter as malloc.
Tfree	free memory allocated by Tmalloc. Requires the same parameter as free.
Trealloc	reuse memory allocated by Tmalloc. Requires the same parameter as realloc.
Tmemcpy	memcpy
Tmemset	memset
Tmemcmp	memcmp
Ttime	time returned in seconds

The following functions are to be coded by the appliance provider. These are prototyped at the end of appl.h and example code exists in appl_sys.c.

Tmustmalloc	memory allocation that MUST be made. No error checking is done by the calling function. If memory cannot be allocated, JetSend core must be shut down.
topsFATALCallback	this is how the device handles shutting down the JetSend core code.

The device may send a message to the control panel notifying the user that JetSend core is shutting down and that the JetSend functionality needs to be restarted (or the system rebooted, etc).

The next section of appl.h is the defines that determine which activity modules are included in the JetSend core compilation. This is clearly defined in appl.h.

In the third section of appl.h there are defines which the device should tailor concerning buffer sizes, the number of connections allowed, etc.

In the final section, appl.h covers platform dependent types. These sections should only be modified by the device if their platform is not adequately represented.

The types defined in this section are:

Tbyte	represents a byte.
Tchar	represents a char.
Tu8	represents an unsigned 8 bit integer.
Ts8	represents a signed 8 bit integer.
Tu16	represents an unsigned 16 bit integer.
Ts16	represents a signed 16 bit integer.
Tu32	represents an unsigned 32 bit integer.
Ts32	represents a signed 32 bit integer.
Tbool	represents an unsigned bit (TRUE or FALSE).

tops.h

The header file, tops.h, provides the appliance provider with an insight to the error values passed between the layers of the JetSend core code. The error code can be parsed and it can be determined where the error occurred as well as particulars about the error (depending on how the JetSend core layer described the error in the code).

Zero is assumed to be good. All non-zero error codes provide information as to an error occurring in the particular layer.

Maks are available to be used by the appliance provider:

TE_GET_LAYER(x) [where “x” is the error code]	which returns the JetSend core layer where the error occurred.
TE_GET_GENERAL_ERROR(x)	which returns the general error from the error code
TE_GET_LAYER_ERROR(x)	which returns the particular error from the JetSend core layer

tops_osp.h

The JetSend core has been characterized as a single layer. It is in fact implemented as three layers, the JetSend Activity Manager (AM), JetSend Interaction Protocol (JIP) layer, and JetSend Session Protocol (JSP) layer. It is necessary to reveal this fine structure of the JetSend core for a very few specific instances. The first thing that is to be done with the JetSend core, and the last thing, are to directly access the JSP layer with associated functions `osp*()`. The `ospSetup()` function must be called before any other JetSend function. Similarly, the `ospCloseDown()` function must be the last thing called. These are defined in the header file tops_osp.h. While there are numerous other routines internal to these JetSend core layers, they need not be called and are not addressed in this ADK.

appl_buf.h

Task block control buffers (TBCBs) are used throughout the JetSend core code when passing data from the AM layer through the TIAPI layer and vice versa. This is defined in the header file, appl_buf.h.

The TBCB structure should be ignored, but the concept of these buffers should be understood. When the buffer is created via a call to `amCreateBuf()` in AM, sufficient room is allocated so that each layer below AM is able to append information onto the buffer. In this way, each layer's remote peer is able to understand something about the buffer going back up the JetSend core stack to AM. Each layer on the remote side is responsible for removing the layer-specific information appended by the sending side prior to sending the buffer to the next higher layer.

Once a buffer is passed to another layer in the JetSend core stack, it is considered to be "owned" by that layer. The owning layer is responsible for either freeing the buffer or passing the buffer on to another layer in order to relieve itself of ownership.

appl_queue.c

Queues are doubly linked lists maintained by appl_queue.c. The queue functions are described in this portion of the ADK. Queues are identified by a TqHead structure. As with buffers, the TqHead structure should be ignored, and only the concept of how the queue functions must be understood.

Basically, queues are created with a call to `QueueInitialize(<TqHead structure>)` and are accessed via the same `TqHead` structure for adding, deleting, parsing elements in the queue. The queues are not restrictive as to the type of information they maintain; that is up to the user. The types of structure, value, etc., are hidden from the queue functions by casting the queue element to a void pointer.

The most important thing to remember with the queue functions is getting the parameters right. If an appliance provider is debugging the queue functions, then obviously something else (wrong parameter, for instance) is the cause of the error.

Chapter 2

JetSend-Enabled Appliance Requirements

This section provides a reference to required resources, both hardware and software, to develop and implement a JetSend appliance. It is broken down into requirements for the appliance itself and also for the development system.

Appliance

Required

- Approximately 90K ROM for the basic, stripped-down JetSend core code. Depending upon your needs, the following must be added to this value to arrive at an accurate required ROM size:

Module	Code Size
Contact	4.5K
Push Receive	2.5K
Push Send	3.5K
Status Receive	4K
Status Send	3.8K
Address Receive	2K
Address Send	2.2K

  See “Negotiation of Surface Information” in Chapter 5 for more information on which modules are required for which types of devices.

- In addition to this you will need ROM for your device-specific code. The amount of additional ROM required depends upon several factors, such as how much code is required to translate your document format to and/or from e-material, and how much you can leverage the existing device network code, if at all. One device implementation required 200K of ROM for both JetSend core and device specific code.

- Access to a timer with a 50ms. resolution and the ability to set a timer to a certain value and receive notification when the timer goes off.
- Access to a network or link supporting unreliable datagram transfer such as UDP over IP and an implementation of the **Reliable Message Transport Protocol (RMTP)**. A reference implementation of RMTP for IP is provided in the ADK. An RMTP implementation is required since connection management is all negotiated at the datagram level; the UDP/IP and RMTP are required for a standard appliance. JSP makes calls into the **transport independent API (TIAPI)**, and that functionality must be supported by your network layer. Please see "JetSend-Enabled Device Architecture" in Chapter 3 for more information.
- Access to the following C standard library functions:

atoi()	memset()	strlen()	tolower()
free()	sprintf()	strncat()	
malloc()	strcat()	strncmp()	
memcmp()	strcmp()	strncpy()	
memcpy()	strcpy()	strpbrk()	

Recommended

A select() type interface into network events is highly recommended. Otherwise, the TIAPI layer will have to be polled about every 50 milliseconds.

Development Hardware

Required

- Personal computer or workstation.
- Access to a network. There is support for a loopback mode under which much development can take place, but eventually access to a network is required.

Recommended

- HP workstation, model 712 or above with *at least* 32 meg RAM

  or

- Pentium-based personal computer with *at least* 16 meg RAM.

Development Software

Required

- C compiler capable of compiling and linking ANSI C.
- Access to a network API such as the Berkeley sockets API for IP.

Recommended

- Windows ’95, Windows NT (4.0 or higher) or HP-UX version 9.x or higher for Windows-based hardware: Microsoft Developer Studio version 4.0 or higherfor Unix systems: the GNU set of developer tools or the standard supplied compiler
- Standard C development tools such as the *nmake* program, a good editor, debugger, etc

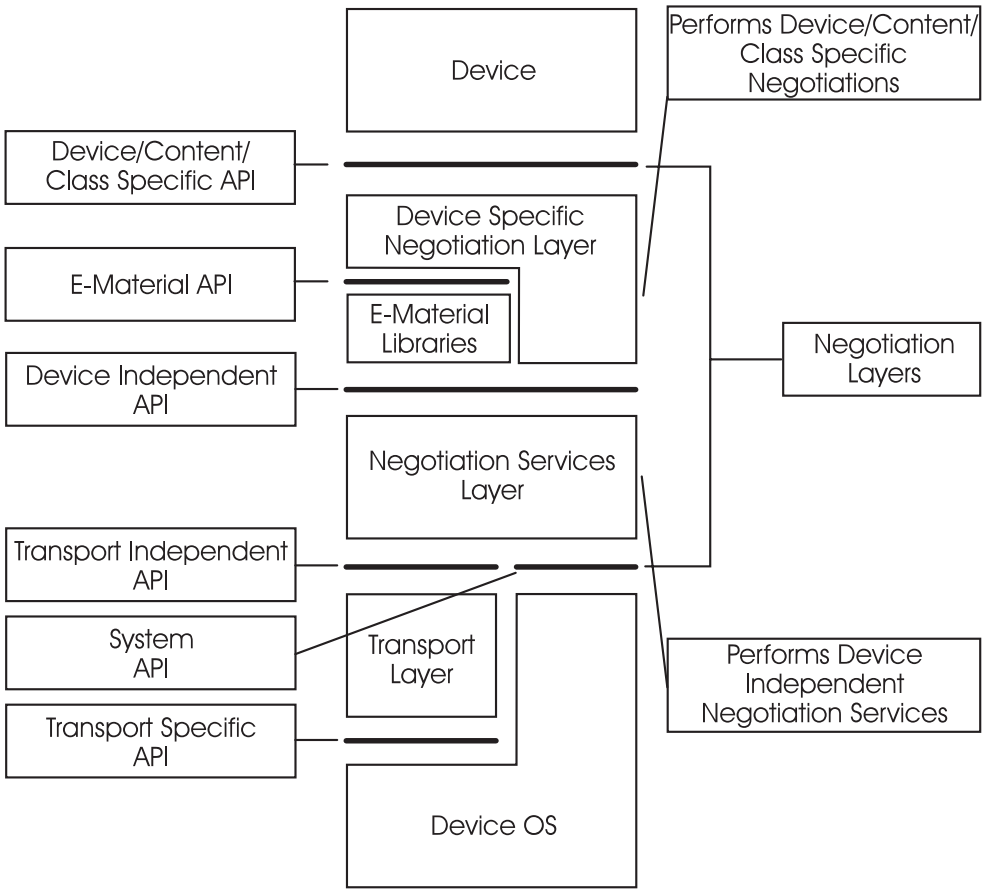
Chapter 3

JetSend-Enabled Device Architecture

Model of JetSend-Enabled Appliance Architecture

A JetSend-enabled appliance automates the transfer of information. The complexity of this operation is encapsulated in the architecture of the JetSend-enabled appliance, which overcomes the complexity by negotiating the information to be transferred.

Figure 3.1 JetSend-Enabled Appliance Architecture



Components of a JetSend-Enabled Appliance

Every computer peripheral can be thought of as a potential JetSend-enabled appliance. Each one has as an element a fundamental device that prints, scans, or performs some other similar function. In addition to the element specific to its operation, it also has an operating system that controls the resources of the device, such as memory, timers, access to network connections, etc. While computer peripherals are not the only candidates to move to JetSend-enabled appliances, they represent a substantial pool from which appliances can be drawn.

To make an existing device a JetSend-enabled appliance, several additional elements must be inserted between the device and the underlying operating system. (This is figurative, since the device can still continue to operate basically as it has in the past.) JetSend inserts two significant layers into the network stack architecture. Thus two appliances, a sender and a receiver, can be described as two network protocol stacks, as shown in Figure 3.2.

Negotiating Layers

The two critical elements that make the peripheral into a JetSend-enabled appliance are the two negotiation layers (see Figure 3.1. The lower layer provides device-independent negotiation services. This JetSend core code has been developed and extensively tested by Hewlett-Packard. These services enable transmission of any type of information through any type of reliable bi-directional connection.

The upper layer provides for the construction, interpretation, and conversion of data, making use of the underlying device-independent negotiation services. This device, class, or content-specific negotiation is specific to a class of data, such as static images, formatted text, or audio-visual clips. Current implementations address static images; other classes may be defined later.

Negotiating done by the upper layer is specific to a particular device. Most often it is convenient to speak about this negotiation in terms of the content being manipulated. For example, a scanner and a printer negotiating together the best format for the transfer of a given static image (or document) both look at the same critical factors; these factors include resolution, compression method, and so forth, hence the name content-specific negotiation. It is, however, occasionally convenient to think and talk either in terms of the devices interacting or the class of materials being handled. Hence content-specific negotiation is equivalent to device-specific negotiation and to class-specific negotiation.

The device-specific code performs the native operations of the device such as the imaging system, storage, and so forth. The device-specific layer does the content-specific negotiations and any required data conversions between the appliance's native format an the on-the-wire e-material format. This is new code that must be developed for each appliance, although it will be similar for most classes of appliances. Definition of content-specific or device-specific APIs is left to individual development efforts. E-material libraries are available for implementing this device-specific code. This API facilitates the construction and interpretation of e-material.

The device-specific code ties into JetSend core through the device independent API, which is generally called simply device API. JetSend core then interfaces with the transport layers through

the transport-independent API (TIAPI). Depending on the selected transport mechanism, the transport layer then ties to the device OS through a transport-specific API.

Negotiation Services

An appliance implements the HP JetSend Communications Technology by integrating the JetSend negotiating services layers, also called JetSend core code, supplied with this ADK.

The negotiation services can handle any type of data defined in the content-specific negotiation layer. A connection is made between two or more JetSend-enabled appliances. Appliances identify themselves, addresses are exchanged, a session is established, one or more jobs are carried out, and status is exchanged. All of these operations are carried out automatically by the negotiation services layer.

JetSend core is designed to be platform- and OS-independent, allowing it to be ported easily into devices with different system architectures. It is possible to implement JetSend core more efficiently on a specific platform in order to take advantage of features like multithreading, signaling, and blocking calls. However, the existing implementation does not use any such mechanisms. It is single-threaded, and events are polled through function calls. JetSend core is written in ANSI C. Although the code has been written for portability, there are still platforms to which JetSend core cannot be ported. [1]

Because the only limitation of the transport layer is bi-directionality, the negotiation services layer also takes care of handling the data at the lower levels. Connection may be made through an unreliable datagram channel that has been made reliable, or through a connection-oriented socket. All of this is also taken care of by the negotiation services layer.

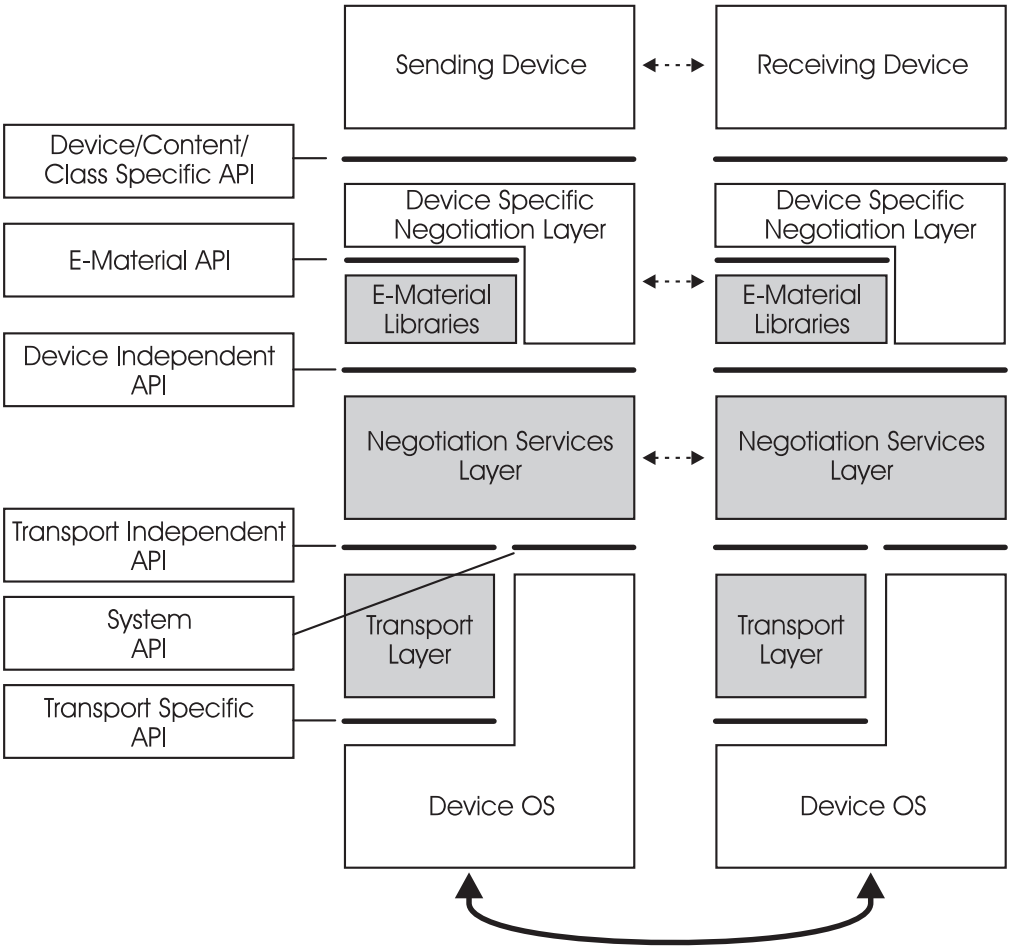
JetSend core ties into the network sub-system and OS through the TIAPI and the system API (SysAPI). The code that translates TIAPI calls to a given network interface on a device is device-specific code. The ADK contains reference code that ties the TIAPI into a Berkeley Sockets interface.

Protocols

Communication between a sending peer appliance and a receiving peer appliance uses a number of nested protocols. As with most networking systems, each layer communicates with an equivalent level in the other device (See Figure 3.2.). Each layer establishes a protocol that encapsulates the data coming from above and is then nested in the protocols of the layers below. So it is with HP JetSend.

1. A Java Virtual Machine, for example, requires JetSend core to be re-written in Java.

Figure 3.2 JetSend-enabled Appliances as Protocol Stacks



The device/content-specific layers (sender and receiver) communicate with each other using what can informally be called a content-specific protocol. In fact, this protocol is established by the sender and receiver, which are constructed using the negotiating services layer.

The negotiation services layers (sender and receiver) communicate with each other using the HP JetSend Core Protocol. This is further broken down into an activities, interaction protocol, and session protocol. This breakdown is generally not apparent above the device independent API. (For a significant exception, see “tops_osp.h” on page 1-17.)

Application Programmatic Interfaces (API)

There are a number of APIs associated with HP JetSend. Six of them might be formalized in a discussion of HP JetSend as a protocol stack with each layer interacting with a corresponding layer in a remote appliance. These are the device specific API, the device independent API , the transport-independent API (TIAPI), and the transport-dependent API. There are two other APIs that would not be viewed as defining layers in a protocol stack, but are essential to the successful implementation of a JetSend-enabled appliance. They are the e-material library API and the system API.

Device Specific API

The device specific API presents the content-specific negotiation functionality to the device. The device, as a part of its normal initialization sequence, must initialize the HP JetSend layers. It directs data out through the API, either as a sending device or as a receiving device that wishes to report status to other appliances. The device also checks for incoming events through the API as appropriate.

The device code interfaces JetSend core through the device independent API, which contains functions to communicate with remote JetSend-enabled appliances. Currently there are no formally defined device specific APIs.

The device specific code implements the negotiation algorithms specific for the given appliance and converts data between the e-material on-the-wire format and the appliance's native format. The ADK provides sample appliance code for simple sending and receiving appliances. Within the device specific code, it will be necessary to call the e-material libraries. These libraries allow the construction and interpretation of e-material.

The device specific API files are supplied by the appliance provider; however, they should be modeled after files in ./dev/common/src and ./dev/common/inc as appl*, and after sample files in ./dev/samples.

E-Material Library API

The e-material library contains functions for composing and decomposing e-material blocks used to manipulate surface descriptions, content requests, and content replies.

The library reads and writes attribute-value pairs into blocks of memory. For a description of the purpose of attribute-value pairs see "JetSend E-Material Specification" in Appendix A. An attribute-value pair consists of an attribute name and an attribute value. Attributes in an e-material block are organized in a tree-like structure where each branch in the "tree" is called a level or a sub-level. The root of the encoding tree is called the null level.

The files are located in ./dev/*/inc and ./dev/*/src as appl_em*. (In the code, em refers to e-material.)

Device Independent API

The device independent API presents the negotiation services layer to the content-specific negotiation layer. Regardless of the particulars of the device, the API will be the same.

Files are in ./dev/common/inc and ./dev/common/src as tops*, am*, oip*, and osp*.

Transport Independent API

The transport independent API presents the functionality of the transport layer to the negotiation services layer. In general, this layer should be a fairly simple veneer with the transport specific API. Files are in ./dev/*/inc and ./dev/*/src as rmtp*, tiapi*, or tops*.

Transport Specific API

The transport specific API is the API made available by the I/O subsystem of the device operating system. Many devices make use of the Berkeley Sockets networking model, so TCP/IP and/or UDP/IP transport mechanisms are used at this level. Other devices may make use of other bi-directional links.

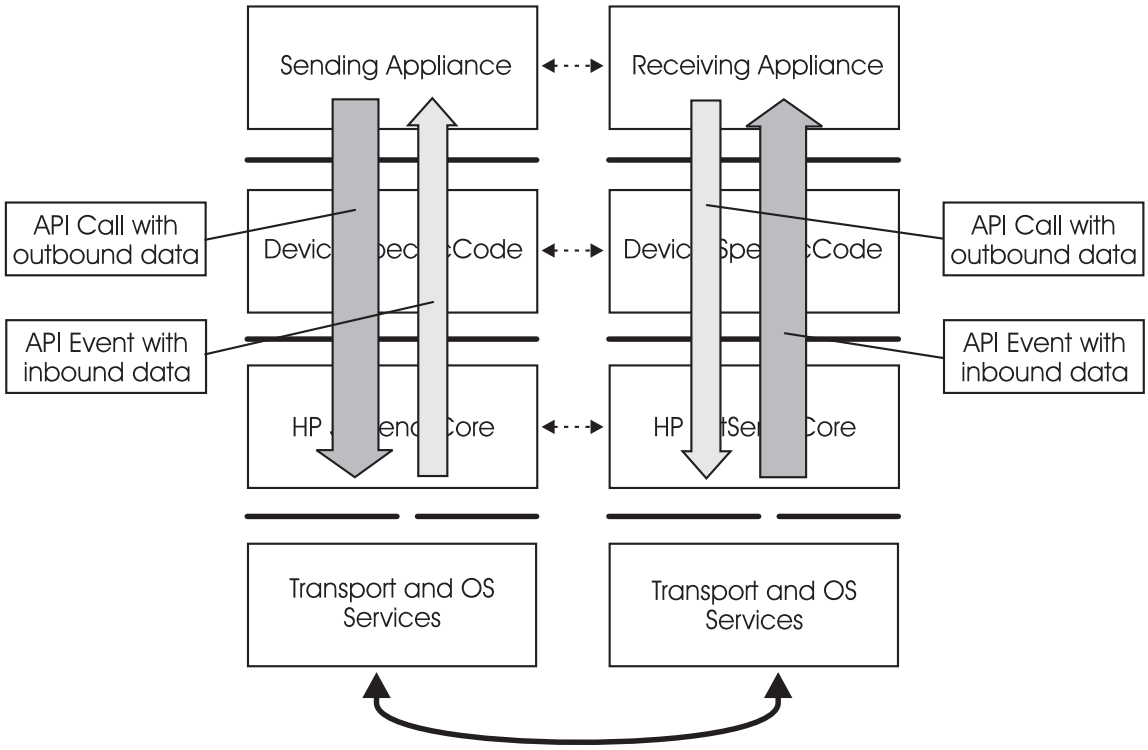
Types of JetSend-Enabled Appliances

Single-Function Appliance

A single-function appliance is one that uses the JetSend-enabled appliance model directly. Applying the principles of JetSend, this appliance is a simple, straightforward, one-function appliance. This appliance will send out a document to another peer or it will receive a document from another peer appliance.

The implementation is quite straightforward as information (in the form of calls) is passed down from the device through the device specific API, device independent API, transport independent API, transport-specific API, device I/O subsystem, and out onto the network. Similarly, information (in the form of events) comes in from the network and is passed up through each of the layers to the device.

Figure 3.3 Data Flow Through JetSend-Enabled Appliances



Hybrid Appliance

An existing peripheral can be combined with a specialized hardware device such as a PC card or printer interface, which together provide this JetSend appliance capability. The existing peripheral connection would roughly approximate the device specific API. The negotiation layers would be implemented within the specialized hardware.

Soft Appliance

Some appliances are implemented using an existing peripheral and a computer to form an appliance equivalent. The various levels of the device I/O subsystem are, in some sense, provided by the computer.

Multifunction Appliance

Some devices have the capacity to perform several functions. A prime example is the multifunction piece of office equipment that provides fax send and receive, copying, printing, and possibly scanning. Such a JetSend-enabled appliance fits within the JetSend architecture. The separate devices are called **logical devices**. The scanning portion would be treated as one logical device and the printing portion as another. Since they share the same device OS and the same I/O subsystem, some parts of the JetSend stack are shared while others are duplicated.

The underlying architecture of a multifunction appliance is non-trivial. Characteristics of this architecture include subaddressing capabilities with multiple logical devices in one physical device. A layer is inserted within the negotiating layers between the JIP and the JSP layers. A single JSP module services mulitple JIP modules. A split stack such as this is not provided in the current ADK.

The PC can be considered the ultimate multifunction appliance, since it is able to support a fair number of separate devices. Each of these devices shares the same IP address but must have unique subaddresses. Some of these logical devices are strictly soft in nature, such as an e-mail gateway. Others are a combination of hardware and software, such as a soft appliance scanner.

Chapter 4

JetSend Communications Model/Requirements

Communication of JetSend-Enabled Appliances

Information flows when two JetSend-enabled appliances interact. Appliances send information, initiate jobs, possibly track status for some time, and end connections.

All interactions between HP JetSend appliances are done through **surfaces**. A surface is an object created on one device that can be viewed by others. A number of operations can be performed on a surface. Surface operations are used for all data transactions between devices, including jobs, status messages, address information, and so forth.

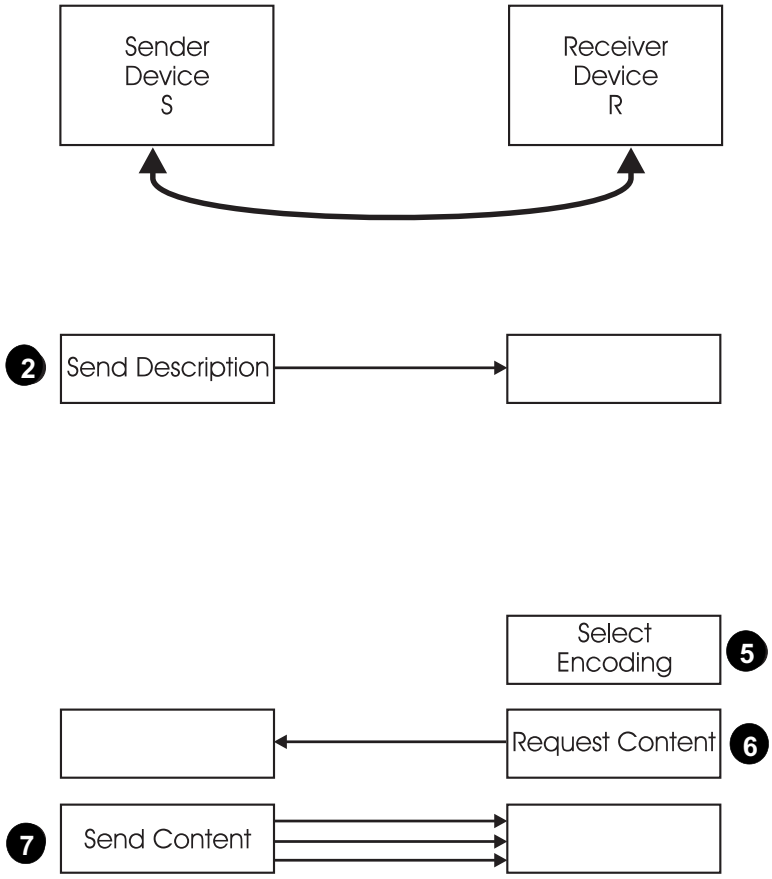
Basic Surface Operations

Surface creation by a sending appliance is called **expression**, and surface receipt by a receiving appliance is called **impression**.

Each surface has a **description** and **content**. These are usually transferred separately. The description is typically small, usually a few Kbytes, the content large, often containing several Mbytes. The description usually indicates several ways of making the content available by the sending appliance. Normally only one of the possible forms is transferred to the receiving appliance. Occasionally, the content portion of the surface is small; in such cases it may be put in-line and transferred along with the description.

Figure 4.1 shows a typical sequence of surface operations between a sender (S) and a receiver (R).

Figure 4.1 Surface Interaction Example



❷ Sender S sends the surface description to receiver R. The description is typically small and contains encoding choices for receiver R as well as general information about the surface content.

❺ Receiver R makes a choice of encoding.

❻ Receiver R issues a content request for the surface. The content is typically large, since it contains actual data for the surface. For example, the pixel data of a static raster image might be the content sent from a scanner to a printer.

❼ Sender S satisfies receiver R's content request by sending the content data.

The surface interaction mechanism, specifically steps ❷ and ❸, is also known as **negotiation**.

Sender S does not have to make the content data for a surface available until step ❼. This allows a sending device to offer a choice of encodings and produce data when it knows the preferred format of the receiver. Sending appliances with limited storage and/or computing capability may make use of this choice to acquire the document from a physical source when the receiving appliance has made a content request.

The receiver controls the negotiation. In step ❺, the description from the sender is interpreted by the receiver using whatever prioities or mechanisms it chooses. The sender is passively listening for requests and satisfying them when they arrive. Either sender or receiver may initiate a transaction, but from that point on the receiver is in charge of the information flow.

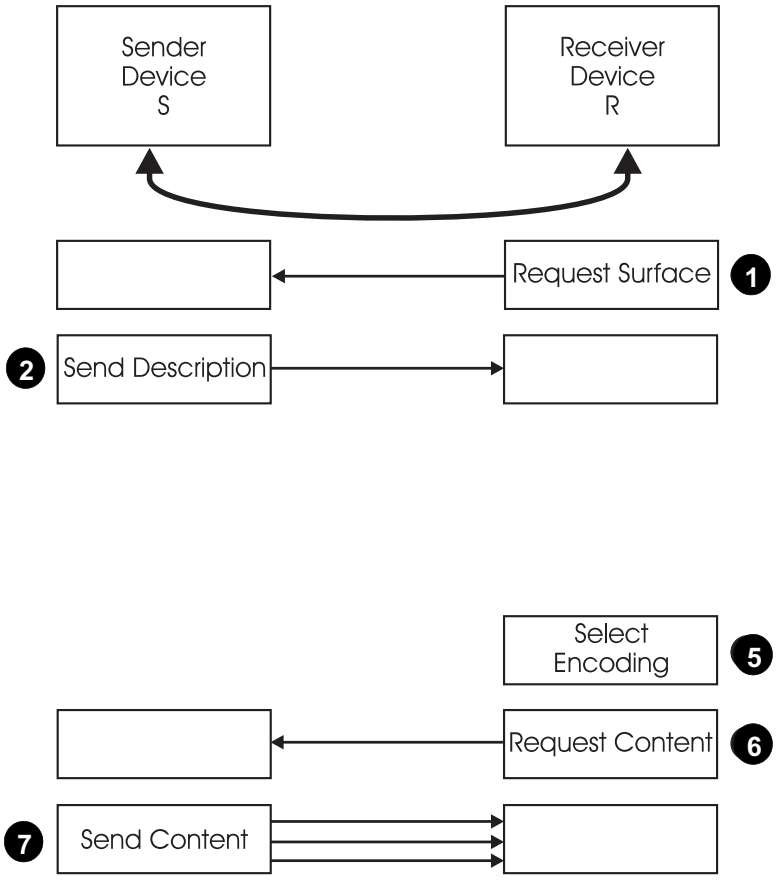
To a large degree, both appliances are simultaneously sender and receiver. Each listens for messages from the other. Each responds with messages to the other. A scanner will listen for and respond to the messages from the printer just as the printer does the scanner. Neither is exclusively sender or receiver.

In the above example, it is assumed that the two devices have already established contact with each other (this defines a **session**).

The receiver or the sender will release the surface at some stage. From the point when the sender sends the surface description to when it is released, the surface is considered live and the receiver can perform operations on it (request content, in this case). When it is released, the sender can free up resources associated with the surface.

Either appliance may initiate a transaction. Senders initiate a transaction by beginning with step ❷—sending a description. Receivers initiate a transaction by beginning with step ❶—making a surface request. (See Figure 4.2.)

Figure 4.2



❶ Receiver R makes a surface request to sender S. Receiver R specifies an ASCII string (a URI) that sender S uses to identify the surface.

❷ Sender S sends the surface description to receiver R. The description is typically small and contains encoding choices for receiver R as well as general information about the surface content.

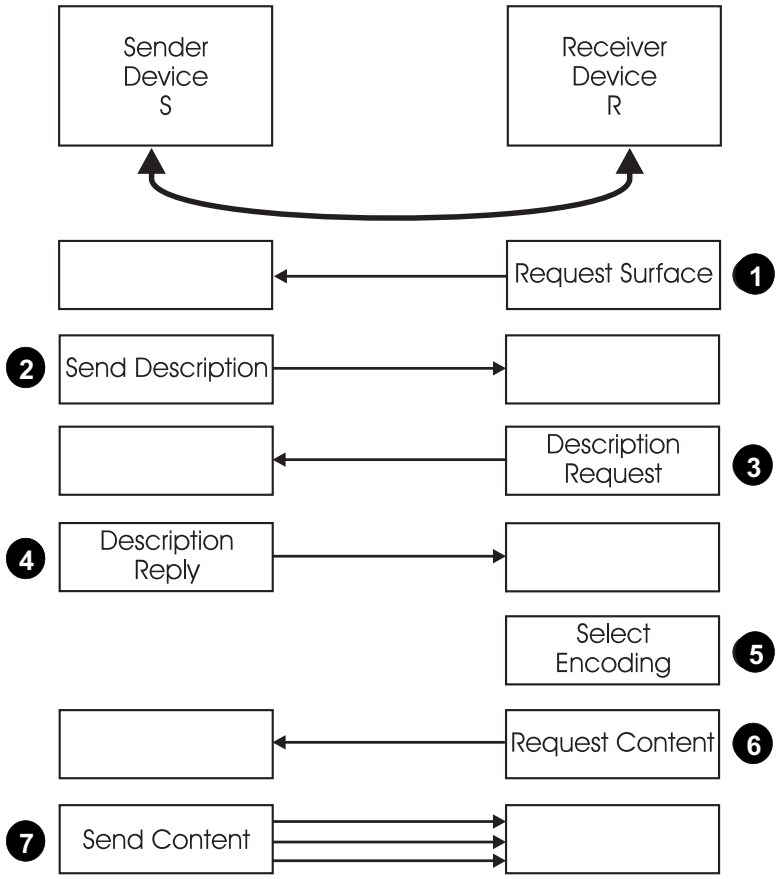
❺ Receiver R makes a choice of encoding

❻ Receiver R issues a content request for the surface. The content is typically large, since it contains actual data for the surface. For example, the pixel data of a static raster image might be the content sent from a scanner to a printer.

❼ Sender S satisfies receiver R's content request by sending the content data.

In Figure 4.2, the receiver requests content for the surface. It may, however, need to request more description about the surface from the sender. This covers the rare case where a surface description cannot fit into one surface message (step ❷) which may be limited to 4K. The description request has a corresponding description reply, so it is similar to, but distinct from, the content request and content reply. The receiver R requests additional description information (step ❸ in Figure 4.3), and sender S completes the description (step ❹).

Figure 4.3



❶ Receiver R makes a surface request to sender S. Receiver R specifies an ASCII string (a URI) that sender S uses to identify the surface.

❷ Sender S sends the surface description to receiver R. The description is typically small and contains encoding choices for receiver R as well as general information about the surface content.

❸ Receiver R recognizes that the description is incomplete and formulates a description request based on the description already received.

❹ Sender S completes the description.

❺ Receiver R makes a choice of encoding

❻ Receiver R issues a content request for the surface. The content is typically large, since it contains actual data for the surface. For example, the pixel data of a static raster image might be the content sent from a scanner to a printer.

❼ Sender S satisfies receiver R's content request by sending the content data.

For example, to initiate the transfer of a complete document, the sender sends the top-level surface of that job to the receiver, which will then request child surfaces of that job and control the process from that point on.

Surface Structure

The surface content is usually the largest volume of information, but the surface description contains the significant characteristics which are used in the JetSend negotiation process. Surface content is what we normally think of as the contents of a document. It may be a bitmap image or a stream of text characters. The surface description encodes the details of the surface content so that the receiving appliance can reconstruct or render the surface as it was originally constructed in the sending appliance.

The surface description is the fundamental building block of surface representation. It describes, using characteristics or attributes, the nature of the surface. This allows JetSend to properly interpret and render the document. (Surface content will be discussed later.)

Surfaces are linked together in a tree-like structure. The structure begins with a surface called the **top-level surface**. All other surfaces are children of this surface. Thus the surface description is recursive, allowing a surface to contain other surfaces.

A surface description can contain references to other surfaces. A surface representing a page, for example, will contain information about the page size, its color and other properties of the page as a whole. For example, it will also contain references (URIs) to subordinate surfaces, or child surfaces such as figures, that are positioned on the page.

The child surfaces are requested separately from the sender. The encoding for each child surface is negotiated independent of its parent surface. So, a page with one image on both sides is transferred using three separate surface negotiations, one for the page and one for each image. A given surface may have more than one parent.

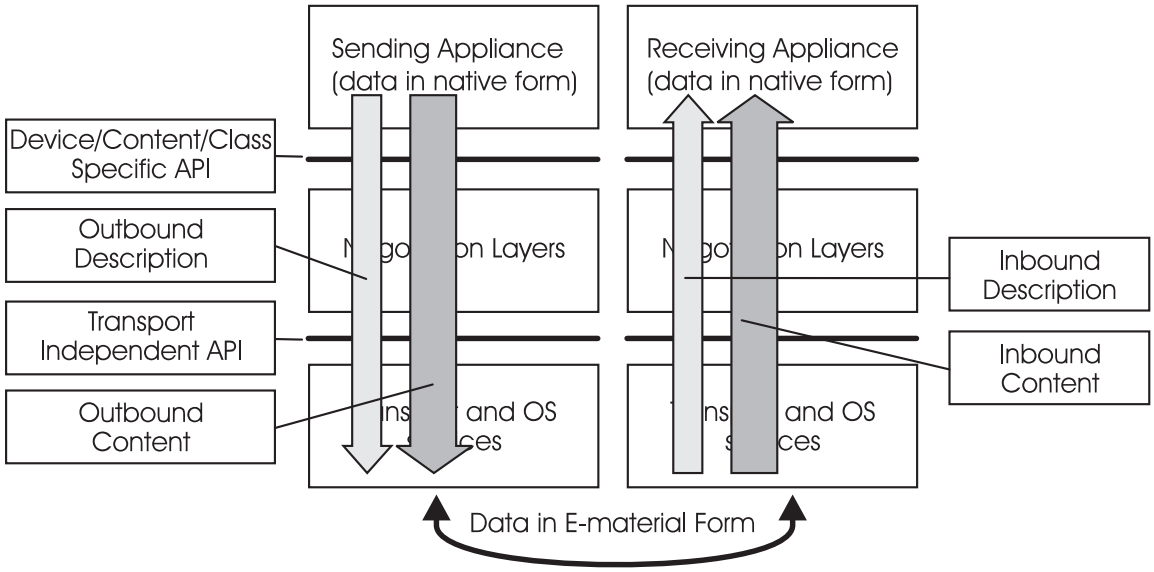
Note that surfaces and child surfaces are often refers to as **job elements**.

Network Exchange View of E-Material

Figure 4.4 illustrates the passing of **surface description** and **surface content** down through the various levels of protocol stacks. Outbound data is passed as **calls** down through the various API levels. Similarly, inbound data is passed as **events** up through the various API levels. The sending appliance typically sends a complete description of a surface. Whether the description is partial or complete, it forms a tree-like structure. If it is not complete, the receiving appliance formulates

and sends **description requests** that are countered by **description replies**. The description request is a list of the attributes (taken from selection lists) along with the selected values in the subtree chosen by the receiving appliance. The sending appliance continues with the description for that subtree. The result of this **negotiation** process is that the receiving appliance will have a **complete description** of the chosen encoding hierarchy's subtree. It will have arrived at the encoding by selecting values from **selection lists** at each level of the process.

Figure 4.4 Network Exchange View



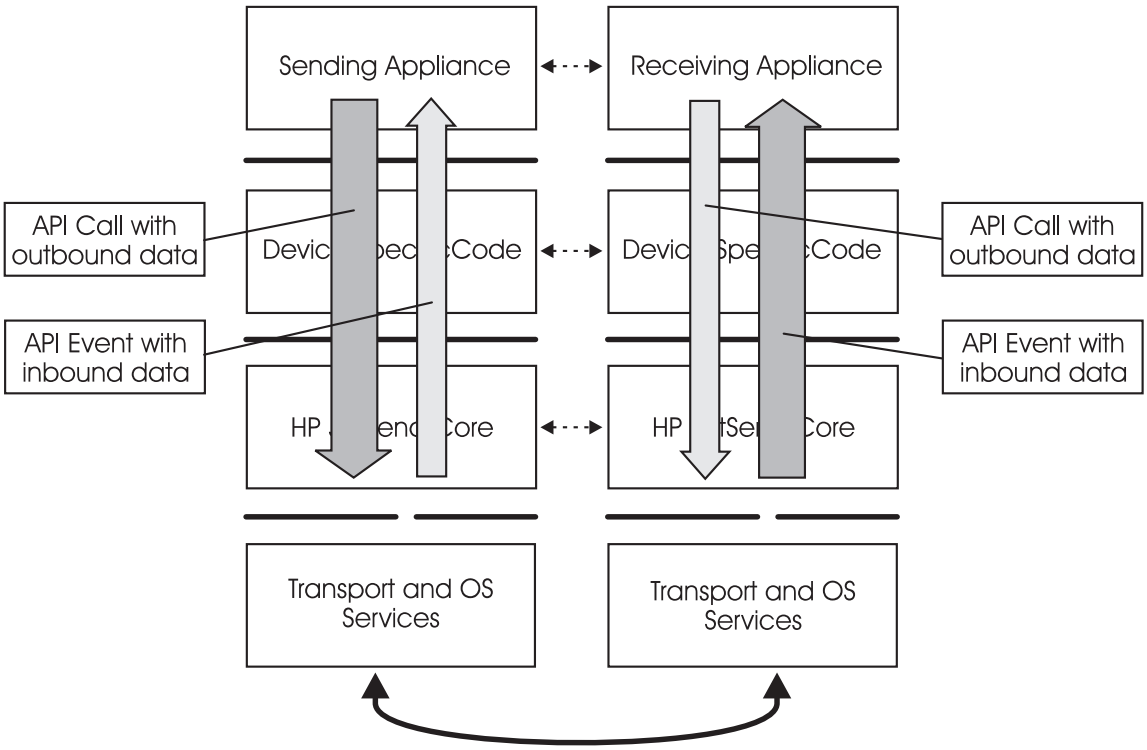
With the required surface encoding decided, a receiving appliance formulates and sends a **content request**. Like the description request, the content request is basically a list of attributes and selection values that the receiving appliance has chosen. In the case of an image, the receiving appliance would specify the color space, pixel depth, resolution, and compression. The sending appliance receives the request and creates the proper **content reply** based on the request. The sending appliance sends the content according to the description. A receiving appliance then receives and renders the contents of the surface.

This process proceeds independently for each surface. It is also independent for each of the several well-known surfaces which together make up a given appliance. Figure 4.5 shows the exchange of one e-material surface. Usually there are many such surface exchanges in the exchange of a document. Several documents may be exchanged during a session; consequently, many messages between sender and receiver are omitted from Figure 4.4 and Figure 4.5.

Buffer Management of Data

The buffer management implementation is critical, but quite straightforward as information in the form of calls is passed down from the device through the device-specific API, device-independent API, transport-independent API, transport-specific API, device I/O subsystem, and out onto the network. Similarly, information (in the form of events) comes in from the network and is passed up through each of the layers to the device.

Figure 4.5 API Calls and Events



Transaction Specifics

Devices use the job modules to send and receive e-material. A job initiated by the sending device is referred to as a push, hence the module names push send (ps) and push receive (pr). A device will have either the sending module or the receiving module or both depending on its function.

Jobs are started by calling psStartJob with an e-material description of the top-level surface of the document. The top-level surface can contain any legal encoding offering (an association, plane, image, or whatever). The receiver will receive a prJobStartEvent with this top-level description, and from then on the receiver controls the negotiation process.

For surfaces that contain child elements (association and plane), the receiver will request the child elements in the order it prefers (prRequestSurface, psSurfaceRequestEvent). The sender should provide the e-material description for child elements being requested (psSendSurface, prSurfaceEvent). For surfaces that have content (image, file etc.), the receiver will request the data (prRequestContent, psContentRequestEvent) and the sender will provide it (psSendContent, prContentEvent).

The sender can reject surface and content requests in case the receiver makes illegal encoding requests (psRejectSurfaceRequest, psRejectContentRequest).

The sending device defines a surface name for every element of a job. An element includes the top-level node of a structure hierarchy as well as each of its descendants (children) of that structure. The surface names are used by the receiving device when it requests new children from the sender. The surface names should be unique for any element of this and other jobs that the sender is dealing with at any one time. It is the device, not the infrastructure, that defines these

names. (They have the type SNAME in the API.) The names are not unique to a receiver end, because a receiver end can communicate with multiple senders using the same names.

Operation Expectations

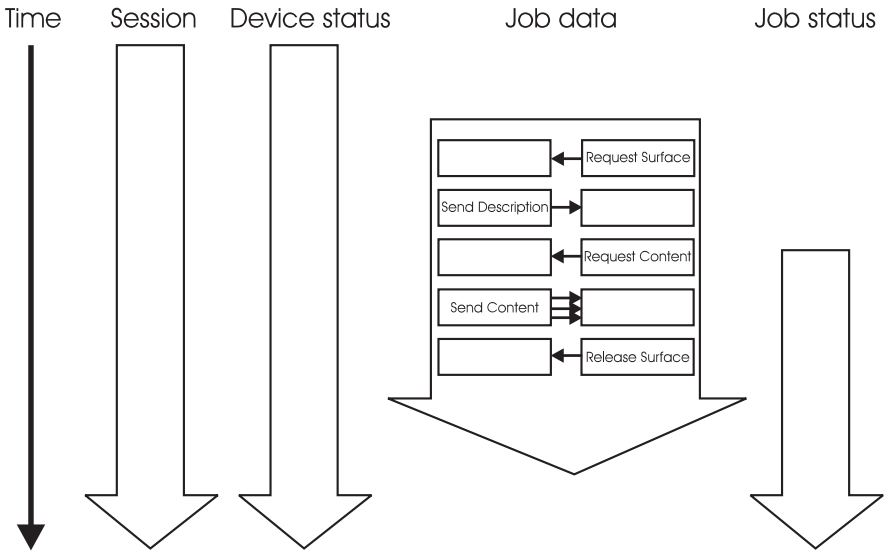
The following operation expectations are handled automatically by the JetSend core code.

Sessions

Two appliances establish a session before they can exchange any information. An appliance can have active sessions with several other appliances at the same time. To start a session, one side listens for incoming sessions and the other side initiates the session. All appliances will typically listen for sessions on all the networks to which they are connected. Sending appliances typically initiate sessions with receiving appliances. Either peer in the interaction can close the session that stops any information exchange until a new session is opened between them. On networks with auto-discovery (such as infrared or direct cable), no device will initiate the session but they will connect if both of them are listening.

Sessions are typically opened for the duration of a job or for monitoring status. Figure 4.6 below shows an open session between two appliances. Each arrow represents the exchange of messages between corresponding well-known surfaces of two appliances. Self surfaces of two appliances are exchanged when a session is initiated. There is a “keep-alive” mechanism used to maintain the session between two appliances, even in the absence of other activity. Device status is exchanged throughout the duration of the session. A job is started some time after the session is established. Job status is provided by the receiving appliance after the job starts, and it continues past the point where all the data for the job has been sent. When the session is closed, no more data is exchanged.

Figure 4.6 Duration of Activities Between JetSend-Enabled Appliances



Channels

Appliances can send data on different **channels** of open sessions. There are currently two types of channels, message channels and stream channels. Appliances must support message channels; stream channels are optional. Appliances are required to have at least a single message channel. It must be a reliable, ordered channel with at least a 4K buffer size. For IP, this is done using RMTP with UDP/IP.

All messages can be sent using the primary message channel. Other channels are negotiated at the network level for any two devices. Two devices that support a stream protocol such as TCP will open a stream for the content data that will be more efficient than the datagram transport. If either side does not support streams, the data will be sent using the message channel.

Most messages, such as status messages and various device information, are sent using a message channel.

Job data or content data is typically sent using a stream channel if one is available. As stated earlier, the content constitutes the bulk of the data transfer, although on the IP network, message channel data is sent using UDP, and stream channel data is sent with TCP. Since UDP messages are unreliable, JetSend implements an **RMTP**. This makes UDP message channel transactions both ordered and reliable.

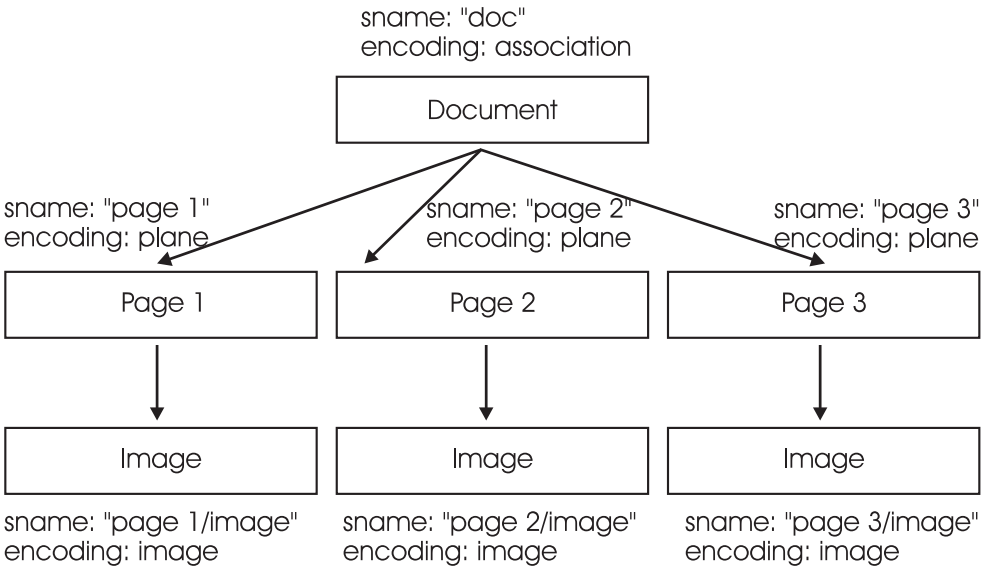
Job Processing

When a device receives some e-material on its *in* surface, it will do its “normal operation.” In principle, this means that it can do whatever it wants but generally a receiving device will render it: A printer will print material it receives, a fax will print it, a PC will store and possibly display it.

The sender cannot control how the receiver processes a job. That means a sender cannot set ‘number of copies’ or ‘use upper tray’ when sending to a printer. And it cannot tell a PC to ‘display it on screen when received’. The receiver can only use properties of the e-material to determine how to process it. For example, a printer can look at the size of the e-material and decide which tray to use if different paper sizes have been loaded.

Figure 4.7 shows a sender transferring a three-page document with raster images on each page. The sender has a model of the document that looks something like this:

Figure 4.7 Example Document Structure Example



Depending on the sender, the **raster image** data for the three pages might reside in separate files on a disk. Alternately, they might be scanned in sequentially through a sheet feed scanner. One very important design consideration for senders is the naming of the job elements (also known as the job surfaces). In Figure 4.7, the sender has named the top-level surface representing the document “doc”, the pages “page1”, “page2”, and “page3”; and the images “page1/image”, “page2/image”, and “page3/image”. It is entirely up to the sender how the naming is done. If the raster images reside in files, a sensible naming scheme might be to name each image by the file name, for example, “c:\tmp\image1.jpg”.

Senders capable of sending more than one job at a time must be able to distinguish surfaces that belong to different jobs by their names. Otherwise, the sender will get requests for the child surfaces of a job without knowing to which job or parent the requested child relates. If this is important to the sender, the surface names should be constructed with that information included.

This sequence of function calls and events will typically take place for transfer.

	Sender (ps)	**Receiver (pr)**	
1	psStartJob (“doc”, emDoc)	prJobStartEvent	emDoc is the e-material description for the document (encoding: association)
2	psSurfaceRequestEvent	prRequestSurface (“page1”)	The receiver got the name “page1” from the child list in emDoc.
3	psSendSurface (emPage1)	prSurfaceEvent	emPage1 is the em description for the first page (encoding: plane)
4	psSurfaceRequestEvent	prRequestSurface (“page1/image”)	The receiver requests the image on the first page (the name was listed in emPage1)

5	psSendSurface (emImage1)	prSurfaceEvent	emImage1 is the em description for the first image which contains an image encoding offering.
6	psContentRequestEvent	prRequestContent (emRequest)	emRequest is a content request for the image data for image1
7	psSendContent (data)	prContentEvent	data is the pixel data for the image in the requested encoding.
8	psSurfaceReleasedEvent	prReleaseSurface (hImage1)	hImage1 is a surface handle from prSurfaceEvent. The image surface is released (the sender can delete resources associated with it).
9	psSurfaceReleasedEvent	prReleaseSurface (hPage1)	The first page is released.
10-17	(step 2-9 repeated for “page2” and “image/page2”)		
17-24	(step 2-9 repeated for “page3” and “image/page3”)		
25	psJobEndEvent	prEndJob	After the three pages and the images on the pages have been successfully received, the receiver ends the job.

The example above shows which function calls are made and in which events they result.

Note that device-specific code must be designed with the asynchronous nature of the JetSend system in mind. Although the sequence of events will probably look like the one above, there are probably going to be status, session and contact messages mixed in with the events shown. Also, the devices can negotiate the job elements in a different order. A sending appliance may be limited as to the order in which job elements are provided. This is covered in “JetSend E-Material Specification” in Appendix A..

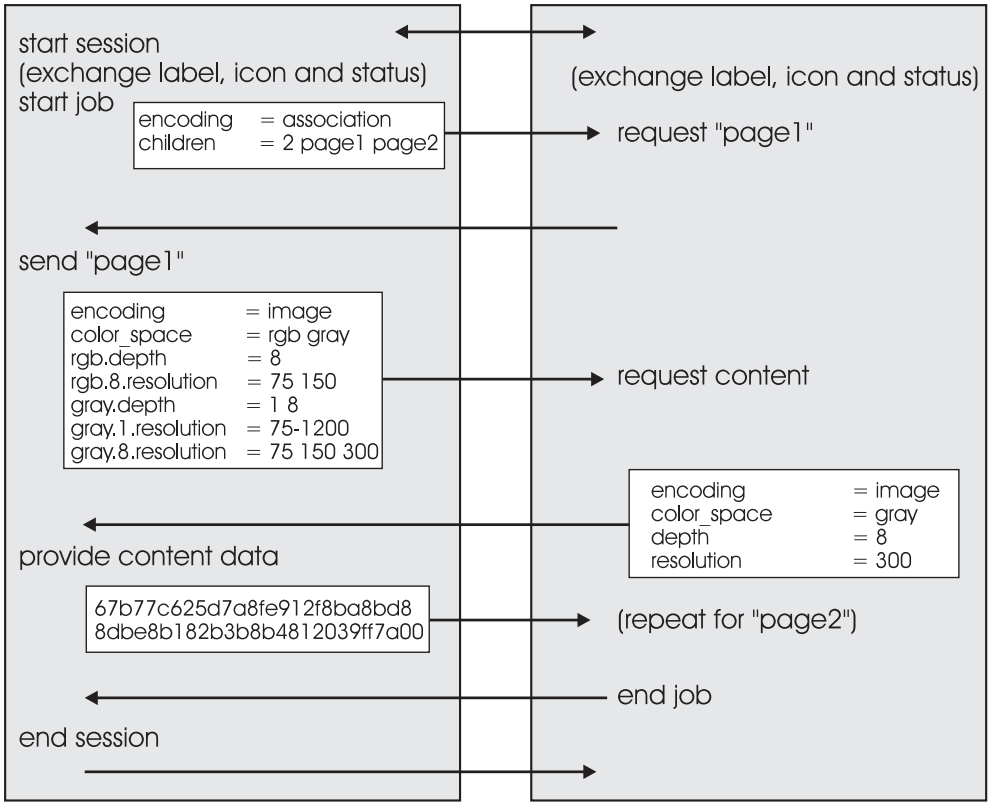
Surface Transaction Example

Each surface of a document is negotiated separately, thus two images in a document can be offered in different encodings. The sending appliance offers a plane with a single image. Each surface of a document is negotiated separately, so two images in a document may be offered in different encodings. Consider the example shown in Figure 4.8. The sending appliance offers a plane with a single image. The diagram shows what information is exchanged between the sender and the receiver.

This mechanism can also be used to re-request the original description for a live surface or to do multi-pass negotiation.

HP JetSend guarantees that content will be available in at least one form. The content appears in a universally accepted form called a **default encoding**.

Figure 4.8 HP JetSend Transaction



Chapter 5

Negotiation of Surface Information

Form of Appliance Communication

Surface Representation

The formal structure of surface messages is called electronic material (e-material). Surface name, surface description, and surface content are integral to the concept of e-material.

The current specification is focused on modeling static images, two-dimensional visual information. It is primarily designed for the electronic representation of information contained on physical paper. So the current e-material specification is used to model documents, pages, sticky notes on pages, and filled regions on pages (raster images or text). In the future, other forms of e-material will be specified.

E-material consists of a set of encodings, each of which represents a separate type of information. The specification of an encoding includes a name and the format of the surface operations that can be performed with the encoding, such as description, content request and reply.

E-material is not a file format. It defines the format of the data that appliances exchange. Internally they will continue to process and store data in their native modes. The JetSend layers handle the necessary conversions. It is the definition of surface descriptions, description requests, description replies, content requests, and content replies. Thus it defines the type of JetSend-enabled appliances can exchange.

Surface Structure

The surface content is usually the largest volume of information, but the surface description contains the significant characteristics which are used in theJetSend negotiation process. Surface content is what we normally think of as the contents of a document. It may be a bitmap image or a stream of text characters. The surface description encodes the details of the surface content so that the receiving appliance can reconstruct or render the surface as it was originally constructed in the sending appliance.

The surface description is the fundamental building block of surface representation. It describes, using characteristics or attributes, the nature of the surface. This allows JetSend to properly interpret and render the document. For the moment, we will ignore the surface content.

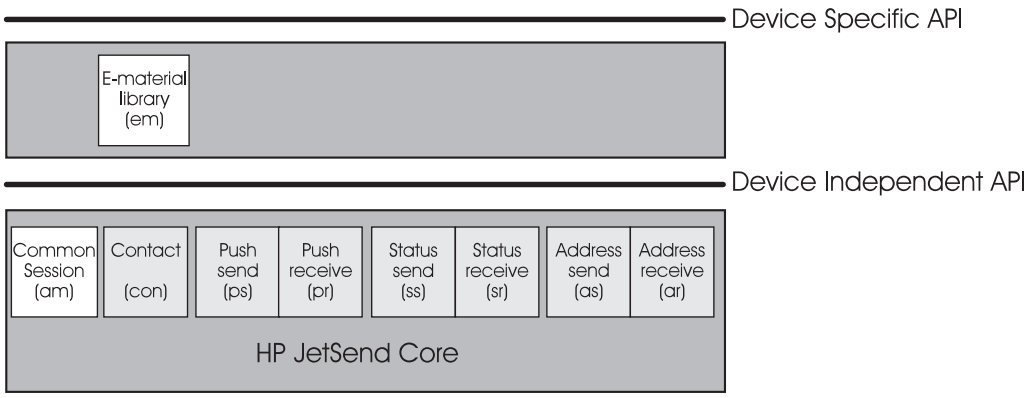
Surfaces can be linked together in a tree-like structure. The structure begins with a surface called the top-level surface. All other surfaces are children of this surface. Thus the surface description is recursive, allowing a surface to contain other surfaces.

Well-Known Surfaces

HP JetSend operates with a set of tagged surfaces, known as well-known surfaces that mean different things when sent to a remote device. Surfaces can be tagged with the following meanings:

- Contact or Self surface—surface contains general information about an appliance, such as its label and icon.
- In surface—surface is the appliance's in-box. When a surface is impressed onto an appliance's in surface, it will process the incoming surface as a job.
- Status surface—surface contains device and job status.
- Address surface—surface contains address information. Address surfaces impressed onto an other appliances will be treated as new address information for the appliance's destination list.

Figure 5.1 ADK APIs and Included Code



Each module contains a number of function calls and events that are prefixed with the letters indicated in parentheses after the module name (am, em, con, etc.). The two white boxes are modules that all devices will use. They contain functions to establish sessions, allocate buffers and manipulate e-material descriptions. The gray modules are called **activities**, and are optional.

Most devices will only need to include a subset of the activity modules. For example, a printer will probably only include the contact, job receiver and status sender activities. It is recommended that all devices include the contact activity that allows devices to exchange names and icons

All appliances will typically have a self surface, allowing users of remote appliances to see the appliance's identity. Receiving appliances will have an in surface where it will receive new jobs from sending appliances. Most appliances will provide a status surface that contains device status, and some will provide job status as well. Only appliances that initiate sessions (mostly senders) will have an address surface that allows them to be remotely programmed.

Table 5.1: Well-Known Surface/Activity Summary

Surface	Activity	Module	Sending Appliance	Receiving Appliance
Self Surface	**Contact**	**Contact**	YES	YES
In Surface	Push Send Push Receive	Job	NO	YES (almost always)
Status Surface	Status Send Status Receive	Status	Usually	Usually
Address Surface	Address Send Address Receive	Address	YES (almost always)	Occasionally

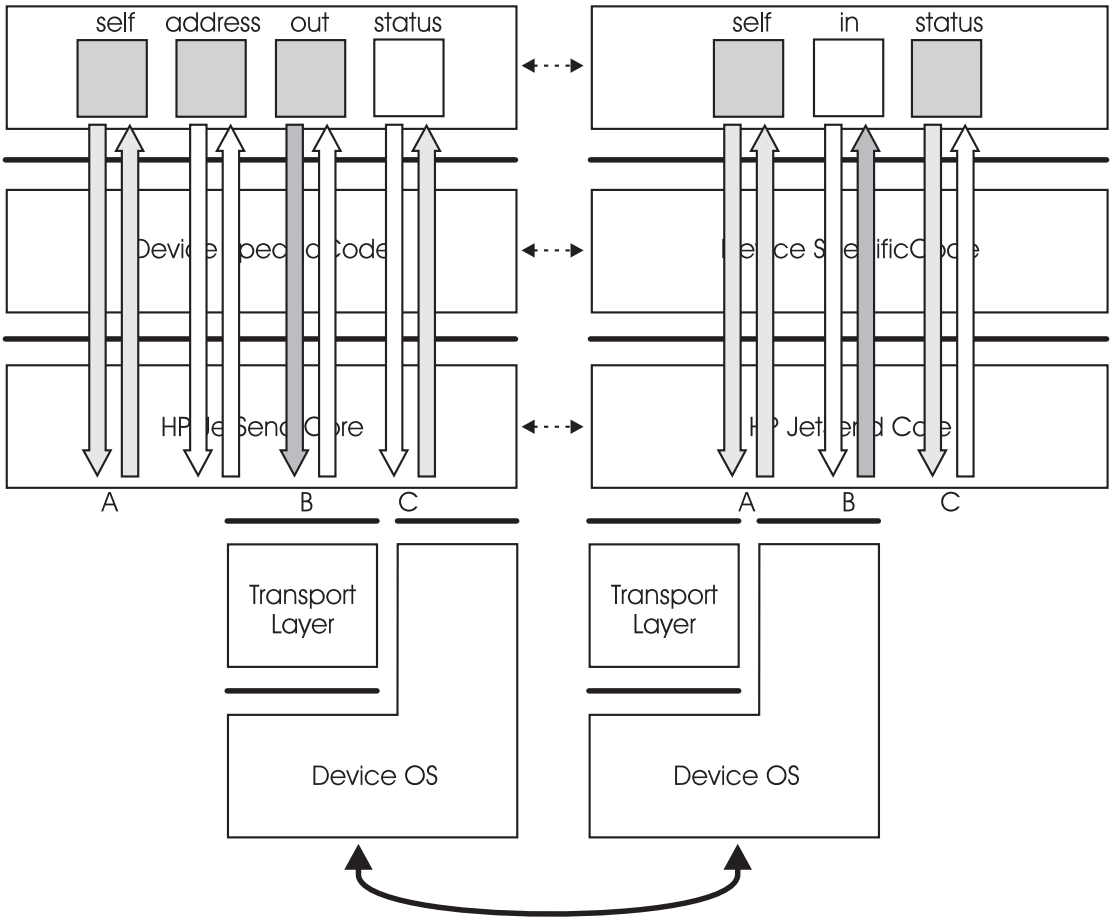
These surfaces are handled individually at the device independent API. Other surfaces may be defined in the future.

Successful transaction between two appliances requires that:

1. Both appliances have a self surface, and one must have an in surface.
2. Both support the same type of information (e-material).
3. Both support the default encodings.

Each well-know surface on the sending appliance exchanges information with a corresponding well-know surface on a receiving device. Figure 5.2 shows a typical pairing of appliances. The sending appliance on the left has a **self** surface, **address** surface, and a **status** surface. It has an additional surface where the data resides in preparation for sending. The receiving device on the right has a **self** surface, an **in** surface, and a **status** surface. You might choose to think of these appliances in terms of a scanner on the left and a printer on the right.

Figure 5.2 Exchanges Between Well-Known Surfaces



The surfaces in Figure 5.2, which are sources of information, are shaded. The self surface on each appliance contains source information about the appliance itself. These sources of information are on both appliances, not just the sending appliance. The address surface of the sending appliance has no corresponding surface in the receiving appliance, but is still a source of information. The status surface on the sending appliance is generally not a source of information; rather, the source of status information is on the receiving appliance.

Although most appliances are both a sending and a receiving appliance, there is still a definite flow of information from where the information *is* to where it *is not*. The arrows in Figure 5.2 that show API calls and events are shaded to indicate the type of data being sent or received. The highest volume and most critical exchange of information is from the **out** surface of the sending appliance to the **in** surface of the receiving appliance.

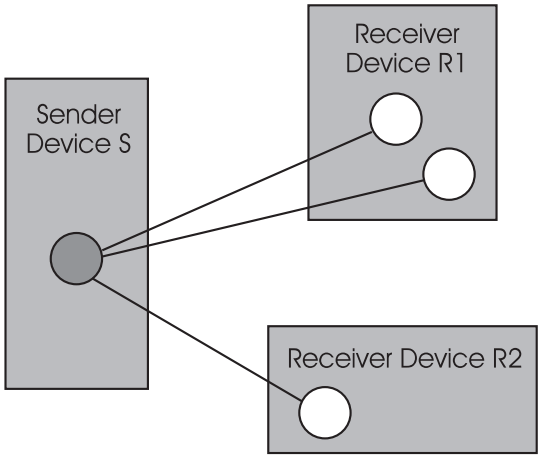
Surface Impression and Surface Change

The protocol allows the same surface to be sent (and kept “alive”) to multiple destinations. In surface lingo, a surface is impressed when its description is (first) sent to a destination. This creates an impression of that surface on the destination. When the surface is released, the impression is removed.

One more surface operation, surface change, completes the protocol for surface interaction.

Surface change is used to change surfaces and their impressions. Consider the following figure:

Figure 5.3 Surface Changes on Sender and Receiver Appliances



A surface is created on sender S; its expression is represented by a black circle on the figure. It is then impressed twice onto receiver R1 and once onto receiver R2. The resulting impressions are marked by white circles. Two types of change operations can be made in this situation:

Sender S issues a change to its surface. This will result in messages to all three impressions. The message contains an updated description for the surface. If the description does not illustrate the change completely, the devices will re-request the content for the surface that changed.

This surface change mechanism is currently used to implement status monitoring between appliances, and can also be used to re-request the original description for a live surface or to do multi-pass negotiation.

A device with an impression (receiver R1 or R2) can request that a change be made to the original surface. Because another device owns the surface, it has to be approved. For example, receiver R2 can change its impression; this sends a change request to sender S. Assuming that sender S approves of the change, change messages will be sent out to all three impressions.

Negotiation between JetSend-Enabled Appliances

The negotiation process draws information from several places. Each element of information influences the process in a different way. The structure of the class arises from the information within the class itself. The particulars of the negotiation tree and preferred encodings arise from each device. The e-material description comes from the document being transmitted, as does the e-material content.

Each device is implemented with appropriate knowledge of the type of e-material class structure of the information that is to be handled. Each characteristic used in the description of the information is known in advance.

In the case of a static image transferable from a scanner to a printer, the possible resolutions of the raster image, the possible numbers of bits per picture element, the possible assignments of the bits to gray scale or color, the possible compression methods must be known, among other things. As the negotiation proceeds, each characteristic is considered, with selections presented by the sender and selected by the receiver.

As seen earlier, a given document may be made available in several encodings. An encoding is formed through the selection of values listed in selection lists. A sending appliance should support a number of encodings, both default and optional. It is not required that a receiving appliance support all encodings.

Obviously, some encodings are more desirable than others. The encodings are ordered by most appliances so that the more desirable encodings are given a preference. This ordering allows the negotiating elements of two appliances—sender and receiver—to identify the encoding preferred by both.

The specific role of a preferred encodings list is to detail the negotiating process. An appliance may use a negotiation tree to control the priority of encodings through the negotiation process. The negotiation tree is a data structure that contains the preferred encodings for the device and places them in order within the tree. The negotiating elements can traverse the tree to find the preferred encodings in order of preference to match them against what is offered. Note that the use of the preferred encodings list may vary among appliances.

The structure of a given document is described using an e-material description. Each surface in the document has a specific place in the structure. The various alternative encodings of each surface are specified in the e-material description.

With the inputs known, the sender and receiver negotiate the best encoding, based on the properties which each appliance possesses.

The receiver, which will render the document, uses the negotiation tree that lists the preferred encodings in priority order. It makes requests of the various surfaces available from the sender. The sender reveals any constraints on the order in which it can send the surface impressions. The receiver has complete latitude within the scope of the e-material class and e-material description to request whatever encodings are required to render the document according to the receiving appliance’s priorities.

Chapter 6

E-Material Fundamentals

Surface Representation

E-material is short for **electronic material**, the formal structure of surface messages. Surface name, surface description, and surface content are integral to the concept of e-material.

The current specification is focused on modeling static, two-dimensional visual information. It is primarily designed for the electronic representation of information that is contained on physical paper. So the current e-material specification is used to model documents, pages, sticky notes on pages, and filled regions on pages (raster images or text). In the future, other forms of e-material may be specified.

E-material consists of a set of encodings, each of which represent a separate type of information. The specification of an encoding includes a name and the format of the surface operations that can be performed with the encoding; this format includes surface operations such as description, and content request and reply.

E-material is not a file format. It defines the form of the data that appliances exchange. Internally they will continue to process and store data in their native forms. The JetSend layers handle the necessary conversions. Some of these layers are provided within this ADK. Others are developed by appliance providers. E-material form lies in the definition of surface descriptions, description requests, description replies, content requests, and content replies. Thus it defines the type of information appliances can exchange.

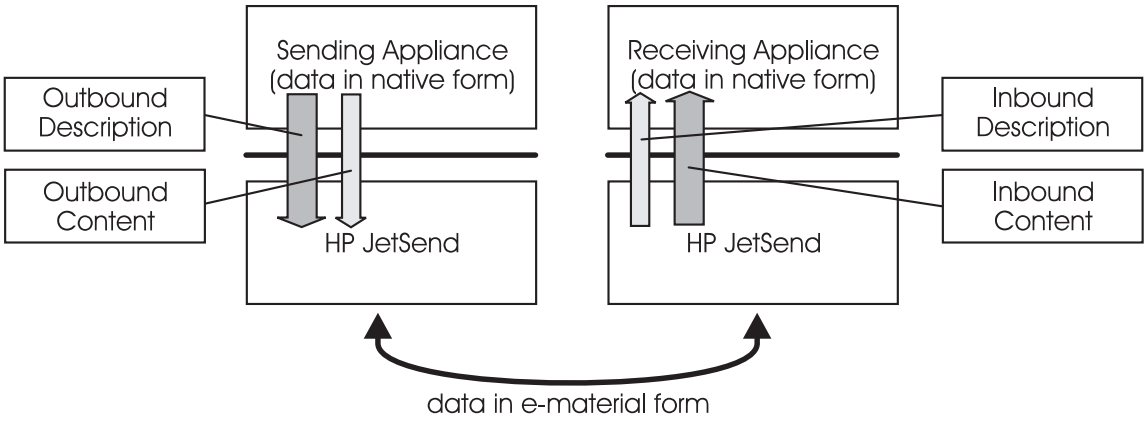
E-Material Exchange

The current specification for e-materials is focused on modeling static, two-dimensional visual information. It is primarily designed for the electronic representation of information contained on physical paper. The current e-material specification is used to model documents, pages, sticky notes on pages, and filled regions on pages (raster images or text). In the future, other forms of e-material will be specified.

A surface is the fundamental building block of e-material. A surface can be thought of as an object that conveys information. This definition is quite loose and will continue to be used loosely. The term surface is applied to the information, the object, the repository in which the information is stored. This usage is not unlike that of a page, whose usage goes from a flat, rectangular white piece of newsprint to the intangible and ever-changing CGI-driven web page.

The formal structure of surfaces is intertwined with and fundamental to e-material. A surface has distinct elements, such as surface name, surface description, and surface content, that make it versatile and useful. Each surface moves from a sending appliance to a receiving appliance in blocks, description and content.

Figure 6.1 E-Material Exchanged by Appliances



Since the surface description is separate from the surface content, the appliances negotiate the best form for the exchange of information. The sending appliance provides a list of available alternative encodings; the receiving appliance selects from the alternative encodings those which best meet its capabilities and requirements. Once the negotiation is complete, the content is exchanged.

E-Material Characteristics

E-material expresses the information content of a surface, and is created/generated/scanned by a sending appliance and is interpreted/accepted/parsed/evaluated/rendered by a receiving appliance.

In a larger sense, a number of sending appliances will create e-material for a number of receiving appliances. Each appliance will have its own characteristics. E-material is capable of conveying the information so that any pair of appliances may maximize the content shared while virtually eliminating the information that each appliance has on its peers.

E-material simultaneously expresses:

- The structure of the document to be sent.
- The representational alternatives for the document.
- The representational capabilities of the sending appliance.

E-material allows transaction efficiency by:

- The sending appliance offerings all representational alternatives for a surface at once. The receiving appliance selects from among the alternatives.
- Large and potentially unnecessary content blocks are sent, when necessary, through separate content requests (requested only if necessary) made by the receiving appliance.
- Small content blocks (which might increase the transaction overhead) are (usually) sent in-line to be used or discarded by the receiving appliance.

E-material has a great deal of flexibility that can be exploited when considering a document:

- A structure exists above the level of an elemental surface. (This is called the surface hierarchy.) Multiple surfaces are combined to create a complete document.
- Structure exists within the elemental level of a surface. (This is called the encoding hierarchy.) Alternative encodings are available for each surface.
- Neither the complete structure nor the complete contents of a document need be known when transmission begins.

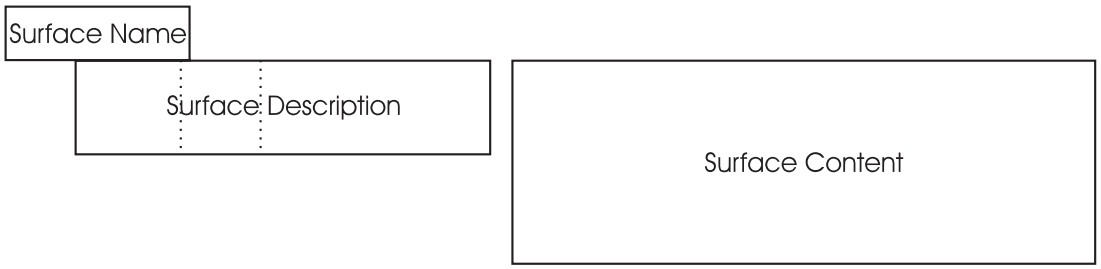
E-Material Surface Structure

An e-material surface consists of a set of one or more encodings, each of which represents a separate offering of the same information. The specification of an encoding includes a name and the format of the surface operations that can be performed with the encoding: description, content request and reply, and so forth.

E-material is not a file format. It defines the format of the data that appliances exchange. Internally they will continue to process and store data in their native modes. The JetSend layers handle the necessary conversions. It is the definition of surface descriptions, description requests, description replies, content requests, and content replies. Thus it defines the type of information appliances can exchange.

A surface consists of a surface name, surface description, and surface content. A surface may be represented in diagram form as in Figure 6.2.

Figure 6.2 Elements of a Surface



The surface content is usually the largest volume of information, but the surface description contains the significant characteristics that are used in the JetSend exchange process. Surface content is what we normally think of as the contents of a document. It may be a bitmap image or a stream of text characters. The surface description encodes the details of the surface content so that the receiving appliance can reconstruct or render the surface as it was originally constructed in the sending appliance.

The surface description is the fundamental building block of surface representation. It describes, using characteristics or attributes, the nature of the surface. This allows JetSend to properly interpret and render the document. For the moment, further discussion of the surface content will be set aside.

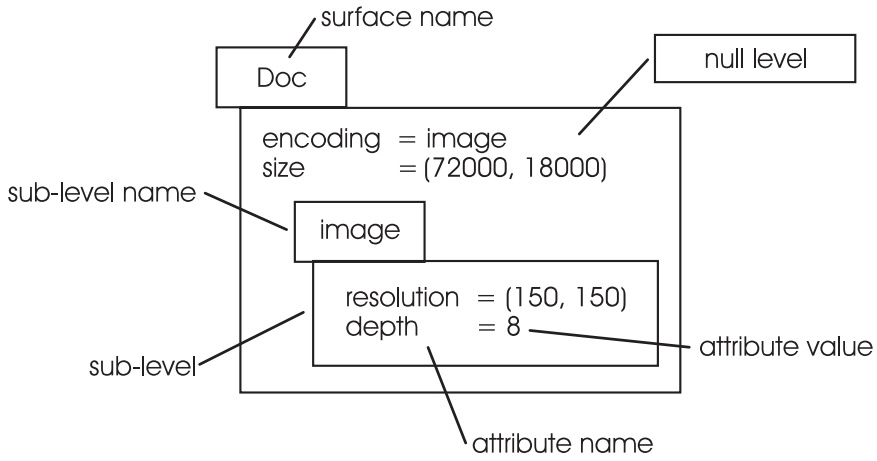
Structure Within a Single Surface

There are several attributes which describe characteristics of the surface. In a particular class of e-material, attributes are given specific precedence or ordering, and are strongly tied to specific values This ordering gives rise to a level or attribute block; the strong binding gives rise to an attribute-value pair. Each attribute-value pair consists of an attribute name tied to a specific value. The attribute-value pairs, along with the name of each level, describe the structure within the surface. At each level there exists one or more attribute-value pairs. The level is named by a decision path.

The structure within a surface is a tree structure. Each branch in the tree is a level or a sub-level. The top level of the tree is called the null level because the decison path is null—no qualification or selection is made on the attributes at that level; instead, the attribute applies to the whole surface. Sub-level attributes apply only to part of the surface.

The figure below shows a simple e-material surface description. The null level contains the two attribute-value pairs and one sublevel. The attribute-value pairs are named ‘encoding’ and ‘size’, and they apply to the whole surface. The value of the ‘encoding’ attribute is ‘image’. The value of the ‘size’ attribute is a pair of integers, 72000 and 18000. The first sub-level is named ‘image’ and has two attributes, ‘resolution’ and ‘depth’. The attribute ‘resolution’ is a pair of integers, 150 and 150.

Figure 6.3 Structure within a Surface



Note that the sublevel name ‘image’ appears first as an attribute value in the null level; this is true of every level name: it will appear as a value in a previous level.:

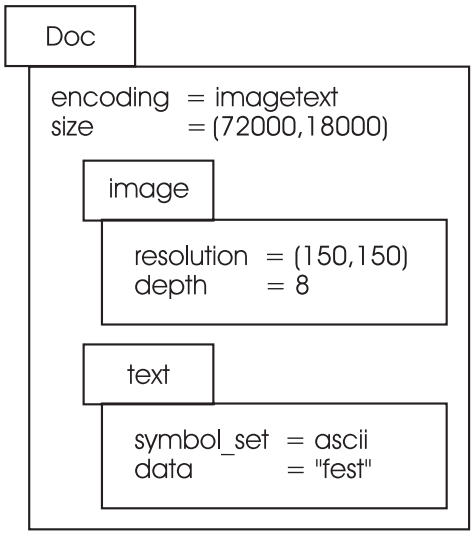
Table 6.1: Surface Description Table for Surface Doc

Level	Attribute	Value
" "	encoding	image
" "	size	(72000,18000)[1]
image	resolution	(150,150)
image	depth	8

[1] Size is based on 72,000 units per inch.

The surface represented in Figure 6.3 can be changed to offer two different forms of the same data—an image and a text string Thus the internal structure of the surface allows for multiple encodings. These multiple encodings are shown in Figure 6.4. A sending appliance offers both encodings. A receiving appliance selects one encoding for rendering.

Figure 6.4 Multiple Encodings in a Single Surface



Again note that the sublevel names “image” and “text” first appear as attribute values in the previous level.

This surface can also be represented as a named table:

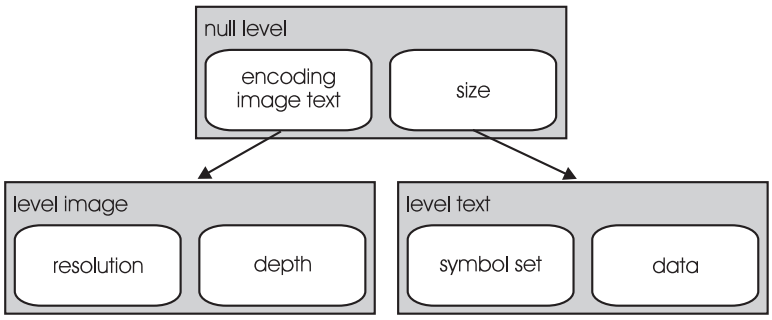
Table 6.2: Extended Surface Description for Surface Doc

Level	Attribute	Value
" "	encoding	image text
" "	size	(72000,18000)
image	resolution	(150,150)
image	depth	8
text	symbol_set	ascii
text	data	“fest”

The attribute-value pairs are qualified by a specific decision path that gives the name of the level. The decision path comes from selected values of attribute-value pairs that appear earlier in the surface description. For this reason, the value portion of the encoding attribute-value pair is called a selection list. The selection list gives rise to the encoding hierarchy.

In Figure 6.5 the same internal surface structure is shown in a hierarchy diagram. All attributes are shown along with those values that become part of a decision path when selected.

Figure 6.5 Encoding Hierarchy of a Surface



As illustrated in these examples, a value can actually have more than one element, depending on the attribute with which it is associated. Some of the values are simple such as a number or string; others are complex such as a pair of coordinates. Some are even more complex.

To insure consistency across JetSend appliances, some encodings are required. These required encodings are called default encodings. Since these encodings are guaranteed to be present, information is certain to be exchanged at some level. If another encoding selection is available, it can also be selected. Alternate choices are called base encodings, or optional encodings. Base encodings are recommended.

To this point, nothing new has been presented about how documents should be electronically encoded. An encoding is an electronic representation of the underlying information in some form.

A three-column textual representation of a surface has been introduced. The first column contains encoding hierarchy information in the form of a decision path that included each level name; the second contains attribute information; the third contains value information related to the attribute. The details of this three-column representation will be discussed subsequently in greater detail.

Structure Linking Surfaces

The structure that links surfaces together is illustrated in the following representation. The attribute “child” is introduced. (There are actually several related attributes that provide this linking.) This attribute takes a reference list as a value. It names other surfaces to which this one is linked. The characteristics relating each child surface to its parent surface are also given, with the level name by the decision path that includes the child surface name.

Table 6.3: Linking Surfaces with Surface Description Table

Level	Attribute	Value
" "	encoding	plane
plane	child	2 sequential A B
plane	size	(8500,11000)
plane.A	position	(250,250)
plane.A	attachment	fixed

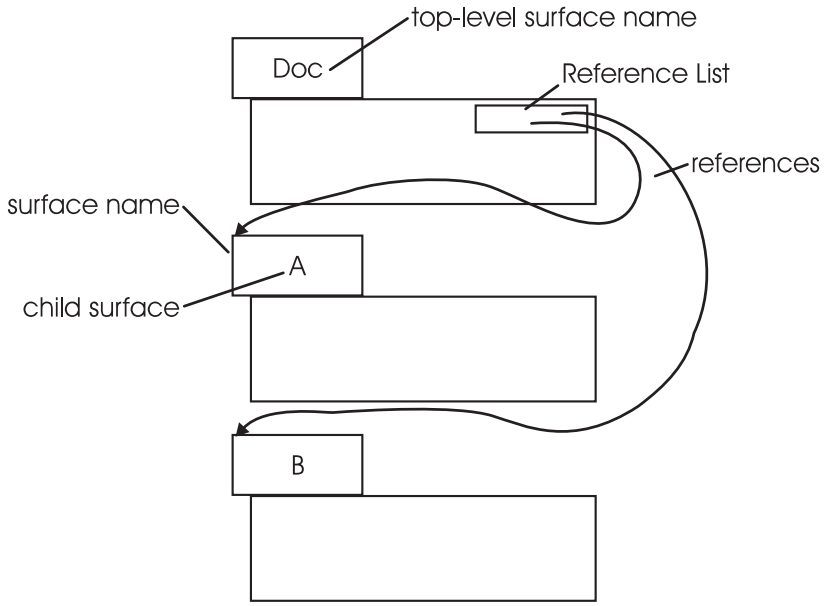
Table 6.3: Linking Surfaces with Surface Description Table

Level	Attribute	Value
plane.B	position	(4000,1000)
plane.B	attachment	float

The “child” attribute takes as a value a list of elements that includes the number of child surfaces it has, what their names are, and how they may be accessed. The figure below shows the same surface linked to the two child surfaces. This is called a surface hierarchy, distinguishing it from the encoding hierarchy.

These two hierarchies are different. When thinking about, talking about, writing about, or sketching a surface, it is essential to distinguish between these two hierarchies. There are times when the two must be represented together; they will be represented differently in these materials.

Figure 6.6 Surface Hierarchy of a Document



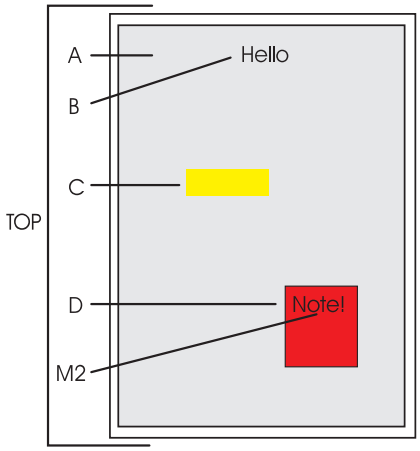
When the surfaces are linked together they form a tree-like structure. The structure begins with a surface called the top-level surface. All other surfaces are children of this surface. Thus the surface description is recursive, allowing a surface to contain other surfaces. Documents may be simple like this one or very complex, with many levels of references. Consider, for example, an ordinary office document. It is in a sense a surface since it is an object which conveys information. It is made up of a number of pages. Each page is, in a similar sense, a separate, distinct surface.

The structure of a full document is expressed in terms of the structure of multiple surfaces linked together and the structure of multiple levels within each surface. Each surface represents an element of the document; each level represents the fine detail within a surface.

E-Material Surface

Consider the surface description of the following document. It consists of a scanned page (shown in grey), with text added at the top, a yellow highlight mid-page, and a red sticky note with text. The physical document is commonly producable. The electronic document would probably be scanned with the latter three elements added after an original scan by a software application.

Figure 6.7 Sample Document



The document represented in Figure 6.7 can be described as a collection of surfaces. Each individual element described above is a separate surface. The document as a whole will be called “Top,” the gray rectangle “A”; the string Hello will be called “B,” the yellow highlight “C,” the red sticky note “D.” The sticky note text will be called “M2.” Using a document description table, this document can be broken down and described. This four-column table gives each element a name, designated in the first column. The structure of each element is further described in the table’s other three columns. The third and fourth columns give an attribute name and an attribute value relating to that attribute. The second column indicates the part of the document to which the attribute-value pair applies.

The field in the value column may be simple or complex. Two complex forms of the value column express the structure of the document. The selection list indicates elements in the second column used in negotiating the transfer of information from a sending appliance to a receiving appliance. The reference list indicates how multiple surfaces can be linked together to form a complete document. In the document description table below, the selection lists are marked with “1” and the reference lists are marked with “2” in Table 6.1 below.

The attribute column contains many characteristics normally associated with images or text, including ColorSpace, Resolution, PixelDepth, Compression, and SymbolSet. Other attributes are somewhat more obscure but not a surprise including Position, Attachment, Opacity, and Size. Each attribute name is encoded by JetSend. To denote these coded key values, they are preceded

by a “v,” and shown emboldened and italicized. Individual attributes and how they are specifically applied comprise a substantial part of the e-material discussion.

Table 6.4: Sample Document Description Table

Surface	Encoding Hierarchy	Attribute	Value
Top	*" "*	***vEncoding***	***vPlane***
Top	***vPlane***	***vChildFront***	4 ***vSequential*** A B C D **2**
Top	***vPlane***	***vSize***	(612000,792000)
Top	***vPlane***.A	***vPosition***	(18000,18000)
Top	***vPlane.***A	***vAttachment***	***vFixed***
Top	***vPlane***.B	***vPosition***	(288000,72000)
Top	***vPlane***.B	***vAttachment***	***vFixed***
Top	***vPlane***.C	***vPosition***	(612000,360000)
Top	***vPlane***.C	***vAttachment***	***vFixed***
Top	***vPlane***.D	***vPosition***	(360000,540000)
Top	***vPlane***.D	***vAttachment***	***vFloating***
A	" "	***vEncoding***	***vImage***
A	***vImage***	***vSize***	(576000,756000)
A	***vImage***	***vOpacity***	255
A	***vImage***	***vColorSpace***	***vGray vSRGB 1***
A	***vImage.vGray***	***vPixelDepth***	1 8 **1**
A	***vImage.vGray***.1	***vResolution***	(300,300)
A	***vImage.vGray***.1.(300,300)	***vPixels***	(2400,3150)
A	***vImage.vGray***.1.(300,300)	***vCompression***	***vNone***
A	***vImage.vGray***.1.(300,300).v***None***	***vEndian***	***vMSB***
A	***vImage.vGray***.8	***vResolution***	(150,150)
A	***vImage.vGray***.8.(150,150)	***vPixels***	(1200,1575)
A	***vImage.vGray***.8.(150,150)	***vCompression***	none
A	***vImage.vGray***.8.(150,150).***vNone***	***vEndian***	***vMSB***
A	***vImage.vSRGB***	***vPixelDepth***	24
A	***vImage.vSRGB.24***	***vResolution***	(150,150)
A	***vImage.vSRGB***.24.(150,150)	***vPixels***	(1200,1575)
A	***vImage.vSRGB***.24.(150,150)	***vCompression***	***vJPEG***

Table 6.4: Sample Document Description Table

Surface	Encoding Hierarchy	Attribute	Value
A	***vImage.vSRGB***.24.(150,150).***vJPEG***	***vEndian***	***vMSB***
B	" "	***vLanguage***	en
B	en	***vEncoding***	***vText***
B	en***.vText***	***vSize***	(72000,18000)
B	en***.vText***	***vSymbolSet***	***vAscii***
B	en***.vText***	***vData***	"Hello"
C	" "	***vEncoding***	***vImage***
C	***vImage***	***vSize***	(72000,18000)
C	***vImage***	***vOpacity***	100
C	***vImage***	***vColorSpace***	***vSRGB***
C	***vImage.vSRGB***	***vPixelDepth***	24
C	***vImage.vSRGB***.24	***vResolution***	(150,150)
C	***vImage.vSRGB***.24.(150,150)	***vPixels***	(150,38)
C	***vImage.vSRGB.***24.(150,150)	***vCompression***	***vJPEG***
C	***vImage.vSRGB***.24.(150,150).***vJPEG***	***vEndian***	***vMSB***
D	" "	***vEncoding***	***vPlane***
D	***vPlane***	***vChildfront***	1 ***vSequential*** M2
D	***vPlane***	***vSize***	(72000,72000)
D	***vPlane***	***vOpacity***	255
D	***vPlane***	***vColor***	(240,20,10)
D	***vPlane***.M2	***vPosition***	(0,0)
D	***vPlane.***M2	***vAttachment***	***vFixed***
M2	" "	***vLanguage***	en
M2	en	***vEncoding***	***vText***
M2	en.***vText***	***vSize***	(72000,72000)
M2	en.***vText***	***vSymbolSet***	***vAscii***
M2	en.***vText***	***vData***	"Note!"

The document description table addresses the description of the document, but says little or nothing about the document's contents. This separation of description and content is important in the utility of JetSend.

Selection lists allow you to encode multiple encodings in a single document description table. Notice that there are selection lists for ***ColorSpace*** and or ***PixelDepth***. This construct allows you to handle the same document in different ways. In the partial document description table below, the image that forms Surface A can be gray scale 1-bit per pixel 300 pixels per inch by 300 pixels per inch uncompressed, gray scale 8-bit per pixel 150 pixels per inch by 150 pixels per inch uncompressed, or standard RGB color 24-bit per pixel 150 pixels per inch by 150 pixels per inch JPEG compressed. These are marked by "1," "2," and "3" in Table 6.2.

Table 6.5: Partial document description table for Surface A

Surface	Encoding Hierarchy	Attribute	Value
A	" "	***vEncoding***	***vImage***
A	***vImage***	***vSize***	(576000,756000)
A	***vImage***	***vOpacity***	255
A	***vImage***	***vColorSpace***	***vGray vSRGB***
A	***vImage.vGray***	***vPixelDepth***	1 8
A	***vImage.vGray.1***	***vResolution***	(300,300)
A	***vImage.vGray***.1.(300,300)	***vPixels***	(2400,3150)
A	***vImage.vGray***.1.(300,300)	***vCompression 2***	***vNone***
A	***vImage.vGray***.1.(300,300).***vNone***	***vEndian***	***MSB***
A	***vImage.vGray***.8	***vResolution***	(150,150)
A	***vImage.vGray.***8.(150,150)	***vPixels***	(1200,1575)
A	***vImage.vGray***.8.(150,150)	***vCompression 2***	***vNone***
A	***vImage.vGray***.8.(150,150).***vNone***	***vEndian***	***vMSB***
A	***vImage.vSRGB***	***vPixelDepth***	24
A	***vImage.vSRGB***.24	***vResolution***	(150,150)
A	***vImage.vSRGB***.24.(150,150)	***vPixels***	(1200,1575)
A	***vImage.vSRGB***.24.(150,150)	***vCompression 3***	***vJPEG***
A	***vImage.vSRGB***.24.(150,150).***vJPEG***	***vEndian***	***vMSB***

Each of these ways of handling the document is called an encoding. Some of these encodings are mandatory. The default encodings insures that two JetSend-enabled appliances can communicate information at some level.

Tabular View of E-Material Surface Descriptions

Thus far, e-material has been represented in several forms, including diagrams of network interchange, tables of document, surface, and level description, and diagrams of surface and encoding hierarchy. Perhaps the most compact and comprehensive model for dealing with the manipulation of e-material surface descriptions is in tabular form.

Attributes, Values, Complex Values

A surface description is specified using attributes and associated values. The attributes of an e-material surface description that are handled by a JetSend appliance are fixed. These attributes are assigned keywords. These will ab designated with a leading ‘v,” and emboldened and italicized.

Each attribute has a value associated with it. The nature of the attribute determines the nature of the value that it takes. Some attributes take on single values—a number or a string, for example. Some attributes take on more complex values. An attribute may allow or require a pair of simple values such as one for a range or a point, a list of simple values, or a list of simple values with some additional structure imposed.

The attributes of e-material surface descriptions are strongly ordered. JetSend imposes a precedence on the various attributes to minimize required processing resources.

Level Description Table

A set of attribute-value pairs may apply to the same specific portion of a surface These pairs are grouped together into what is called an attribute block, or level. This block is placed in a two-column table called a level description table. The first column of the two-column table gives the attribute. The second gives the value field. This may be a simple number or string. It may be substantially more complex. These columns are known formally as:

- attribute (short for attribute name)
- value (short for attribute value)

Table 6.6: Level Description Table

Attribute	Value
vSize	(576000,756000)
vColorSpace	***vSRGB***
vOpacity	255

Selection List, Decision Path

The selection list is used to link together various level description tables. Each level description table is named using a decision path. The decision path is constructed from values taken from selection lists at higher levels.

The decision path labels an attribute according to a tree structure. The highest level has a null decision path and is called either the null level or the root attribute block. Since the decision path qualifies the context of the attribute-value pairs in the attribute block, the null decision path labels those attribute-value pairs that apply to the whole surface.

Every decision path is constructed from selectable elements of the selection lists and reference lists. Similarly all selectable elements of any selection list or reference list must appear in one or more decision paths of a complete surface description. Through the selection lists, the sending appliance offers its available encodings. The receiving appliance selects from those encodings the one(s) that best fit its capabilities and/or requirements.

The order in which the attributes appear in the table and in the e-material class definition is the order in which the corresponding selection values appear in the decision path.

Surface Description Table

A surface is completely described using a surface description table. This three-column table is constructed from a set of level description tables. The first column contains the level name in the form of a decision path. The second and third columns contain the original columns of each level description table. These columns are known formally as:

- encoding hierarchy
- attribute
- value

The levels are ordered in depth-first order according to the level name expressed in the decision path. Other orderings are not allowed.

A surface definition table embeds each of the several level description tables in the last two columns as shown.

Table 6.7: Surface Definition Table

Encoding Hierarchy	Attribute	Value
" "	***vEncoding***	***vImage***
vImage	***vSize***	(576000,756000)
vImage	***vOpacity***	255
vImage	***vColorSpace***	***vSRGB***
vImage.vSRGB	***vPixelDepth***	24
vImage.vSRGB.24	***vResolution***	(150,150)
vImage.vSRGB.24.(150,150)	***vPixels***	(1200,1575)
vImage.vSRGB.24.(150,150)	***vCompression***	***vJPEG***
vImage.vSRGB.24.(150,150).***vJPEG***	***vEndian***	***vMSB***

Reference List, Surface Name

The reference list is used to link together surface by referencing various surface description tables. Reference lists are associated with a small number of attributes such as ***vChild*** or ***vChildFront***. Each surface description table is given a unique name that is an ASCII string.

Document Description Table

A document is completely described using a document description table. This four-column table is constructed from a set of surface description tables. The first column contains the surface name. The second, third, and fourth columns contain the original columns of each surface description table. These columns are known formally as:

- surface hierarchy
- encoding hierarchy
- attribute
- value

The surfaces are ordered in a manner that resolves any existing overlaps. There may be other constraints on the order in which surfaces may be offered.

A document description table embeds each of the several surface definition tables in the last three columns as shown.

Table 6.8: Document Description Table

Surface	Encoding Hierarchy	Attribute	Value
Top	" "	***vEncoding***	***vPlane***
Top	***vPlane***	***vChildFront***	2 ***vSequential*** A B
Top	***vPlane***	***vSize***	(612000,792000)
Top	***vPlane***.A	***vPosition***	(18000,18000)
Top	***vPlane***.A	***vAttachment***	***vFixed***
Top	***vPlane.***B	***vPosition***	(288000,72000)
Top	***vPlane.B***	***vAttachment***	***vFloat***
A	" "	***vEncoding***	***vImage***
A	***vImage***	***vSize***	(576000,756000)
A	***vImage***	***vOpacity***	255
A	***vImage***	***vColorSpace***	***vSRGB***
A	***vImage.vSRGB***	***vPixelDepth***	24
A	***vImage.vSRGB***.24	***vResolution***	(150,150)

Table 6.8: Document Description Table

Surface	Encoding Hierarchy	Attribute	Value
A	***vImage.vSRGB.***24.(150,150)	***vPixels***	(1200,1575)
A	***vImage.vSRGB***.24.(150,150)	***vCompression***	***vJPEG***
A	***vImage.vSRGB***.24.(150,150) .***vJPEG***	***vEndian***	***vMSB***
B	" "	***vLanguage***	“en”
B	en	***vEncoding***	***vText***
B	en.***vText***	***vSize***	(72000,18000)
B	en.***vText***	***vSymbolSet***	***vAscii***
B	en.***vText***	***vData***	“Hello”

Top-Level Encodings

Every value appearing in a selection list and subsequently in a decision path is part of an encoding. There may be as many as seven selection lists in a given surface description or as few as one, each with one or more selectable values. Each selectable value contributes to an encoding. For example, an image may have ***vLanguage***, ***vEncoding***, ***vColorSpace***, ***vResolution***, ***vPixelDepth***, ***vCompression***, and ***vEndian*** as attributes that have a selection list of selectable values. An encoding is a set of specific values for each of these attributes. In the table above, the decision path of the encoding shown is ***vImage.vSRGB***.24.(150,150).***vJPEG*** for surface A. Selection of values for the ***vLanguage*** and ***vEndian*** attributes would extend this decision path even farther.

As implemented, every attribute that takes a selection list and every selectable value within a selection list is essentially equal. The only exception would be when the first selectable value is used as a default. Each contributes to what the encoding is. While all attributes that take selection lists are essentially equal, the vEncoding attribute is discussed specially here. The attribute is called a top-level encoding even though vLanguage appears at the top of the attribute precedence list. The selectable values of this attribute are particularly important in the operation of the JetSend. There are five selectable values addressing different types of e-material:

- ***vImage***—a static raster image. Negotiable attributes for image data include resolution, color space, color depth, pixel encoding, compression, and so forth.
- ***vText***—a simple text string of ASCII or Unicode text that fills a rectangular plane with no font or spacing information. A text string can be offered in different languages for localization purposes. The simple text encoding is used mainly for status information, but can be used to annotate pages as well.
- ***vFile***—essentially a binary file or, more simply, a sequence of bytes. A file name, mime-type, and an icon can be associated with the data. This encoding is different than the ones above in that is does not necessarily contain visual information. Computing devices are most likely to use this encoding.

- ***vPlane***—a two-dimensional rectangular plane with a background color and an ordered "layer" of child elements on its front and back. No attributes of planes are negotiated. A page is a plane containing other encodings such as images or text.
- ***vAssociation***—an ordered sequence of child elements. There are no negotiable parameters for associations. A document is an association of pages.

These five top-level encodings present different ways in which a single piece of information can be presented. Each one has situations in which is performs better, but the mapping of information onto these encodings presents substantial overlap. For example, the information on a piece of paper could be a ***vPlane*** with one vImage on the from and one ***vImage*** on the back; it could be a ***vPlane*** with several vImage and ***vText*** blocks on the front and several others on the back; it could also be an assocation of two images, downplaying the notion of front and back; or, although not recommended, it could be a file containing some application-specific page description language. None of this has addressed the information content of the piece of paper.

Default and Base Encodings

The e-material concept requires that two appliances exchange information at some level, regardless of their inherent characteristics. In order to do this, every appliance provides some universal and fundamental encodings of the information. These are called default encodings. An e-material class has a set of required attributes. There is an ordering or precedence to this set. (It may also have optional attributes; each of these has an order position within the set.) Many of the attributes take as a value a selection list. Among the selectable values some are required, some optional. All of these selectable values will appear in decision paths in subsequent levels of the e-material. Additionally, some selectable values—called base encodings—are strongly recommended. They raise the level of the information common between two appliances. Other, optional encodings are also encouraged.

Currently, only image has default encodings because the other encodings do not have negotiated parameters. **The default encoding for image is 300x300 dpi, monochrome, RLE compressed raster.** That means that all JetSend image devices (printers, scanners, copiers, cameras, whiteboards, others) must be able to send and/or receive this encoding. Similarly, images are recommended to be sent and/or received in the color base encoding.

Code-Level View of E-Material

Summary of Views

Thus far, e-material has been discussed mostly in terms of a tabular representation, in the form of surface description tables. We have also seen schematic representations as a surface hierarchy diagram, an encoding hierarchy, and blocks of attributes with single or multiple encodings. There are other ways in which e-material is represented, always keeping in mind that e-material only exists "on the wire." The real e-material is constructed with calls through an API, interpreted with calls through an API, and transferred between appliances over a transport medium using specific protocols.

E-Material Library

A surface description table gives the description of a surface. Each line of a properly constructed table gives a set of information passed out through the API of the sending appliance and arriving through the API of the receiving appliance. Roughly stated, each line of a surface description file corresponds to a procedure call in the e-material library. One peer passes information out through a procedure call, the other receives information through an event. The sending and receiving appliances construct and interpret the e-material according to the structure specified in this document.

E-Material Type

The procedure calls in the e-material library API consistantly map the three columns in the surface description table into three of several parameters.

- The first parameter relates to the encoding hierarchy, and must be of type emLevelType. Levels are added one at a time using a pointer to the previous value and a selected value. The selected value may be of any simple type listed in the table below. This excludes `emLevelType, emListType`, and `emDataType`. The full decision path as described in the surface description table is never formed explicitly.
- The second parameter gives the attribute a name. This name is generally either an enumerated value of type `emKeyType`, or in certain circumstances a string of type `emStringType`. The attribute name may be any simple type listed below.
- The third parameter gives the value from the surface description table. This can be almost anything from a simple value, to a complex value, to a selection list, to a reference list, depending on the attribute. The value parameter may be any type listed in the table below except `emLevelType`.

Table 6.9: E-Material Data Types

EMtype	Description	Example
emLevelType	level	***vImage*.*vGray*.8**
emKeyType	keyword (16-bit integer)	***vImage***
emStringType	ASCII string	“fest”
emIntType	32-bit signed integer	8
emIntPairType	pair of integers	(100,200)
emRangeType	range of integers	100..600
emRangePairType	pair of integer ranges	(100..600, 200..600)
emListType	list of values	[2 “fest” ***vImage*** -12]
emDataType	binary data	0x20 0x45 0x00 0xf2 ...

The e-material API defines functions that manipulate these EMtype values interchangeably. To do this, a structure is defined that can hold any of these values. Most of the functions to create e-material take a parameter of the EMvalue type. This structure contains an EMtype field specifying the type of value and the data for the value in a union of fields that is specific to each type. So, a variable of the type EMvalue can contain either an integer, a string, a range, a list, or any other type. The library defines EMvalue as follows:

```
typedef struct _EMvalue {
     EMtype    iType;
     union {
        Tu16    key;
        char*   string;
        Ts32    sint;
        EMpair  range, intPair, rangePair[2];
        struct { Tu32 iSize; Tbyte* pData; } data;
        struct { Tu16 iNumber; struct _EMvalue* pValues;} list;
     } v;
} EMvalue;
```

Creating a Surface Description

The e-material library contains functions for composing/creating and decomposing/ interpreting e-material blocks used in surface descriptions, description requests, description replies, content requests, and content replies. A sending appliance constructs a structural description of the surface or surfaces to send. This may be known completely or partially. Once the structural description of a surface is defined, code places it in an e-material block in preparation for transmission. Given a surface description table with standard order, translation to a sequence of e-material library calls is quite straightforward. Consider the surface description table below:

Table 6.10: Surface Unnamed

Encoding Hierarchy	Attribute	Value
" "	***vEncoding***	***vImage vText***
" "	***vSize***	(100,200)
vImage	***vResolution***	(100..600)
vImage	***vDepth***	8
vText	***vSymbolSet***	***vAscii***
vText	***vData***	“fest”

This information is transcribed into function calls using the basic functions `emPrepare`, `emAdd`, and `emAddLevel`, as well as two conversion functions: `emList`, `emIntPair`. The

following code will create the surface description block corresponding to the surface description table above.

```
Tbyte*   em;
Tbyte*   level;
em = Tmalloc(2048);
emPrepare(em, 2048);
emAdd(em, NULL, vEncoding, emList(2, vImage, vText));
emAdd(em, NULL, vSize, emIntPair(100,200));
level = emAddLevel(em, NULL, vImage);
emAdd(em, level, vResolution, emRange(100, 600));
emAdd(em, level, vDepth, emInt(8));
level = emAddLevel(em, NULL, vText);
emAdd(em, level, vSymbolSet, vAscii);
emAdd(em, level, vData, emString("fest"));
```

First, a block is allocated and prepared as an e-material block (em). Second, two attributes are added to the null level with `emAdd()` function calls. Third, the two sublevels are added one at a time. The `emAddLevel()` call returns a pointer to the level used in subsequent emAdd() calls for attributes belonging to that level.

The transcription from surface description table to code requires that the table be in standard order. Other orderings will cause failures, because attribute-value pairs cannot be added correctly. The standard order for entries in the encoding hierarchy is depth-first. When the surface description is transcribed into operating code, the standard order is required to prevent attribute insertion failures in current and future JetSend offerings.

Actual implementation will likely be of a somewhat different style. However, this shows that with the surface description, the coding is straightforward.

Interpreting a Surface Description

A receiving appliance interprets a surface description. Based on the available description, it makes content requests and renders the document.

A given type or class of e-material has a well defined structure with required and optional attributes in a prescribed order. Beginning with the null level of the encoding hierarchy and the top-level surface of the surface hierarchy, the receiving appliance probes the surface description for encoding details. The receiving appliance will probe for every attribute in the class definition. This is done using three basic function calls: `emGetLevel()`, `emGet()`, `emFreeValue()`. The function `emGet()` returns an EMvalue variable.

```
EMvalue value;
level = emGetLevel(em, NULL, vImage);
if (level && emGet(em, level, vDepth, &value))
{
     if (value.iType == emIntType)
```

```
        printf("The depth is %d\n", value.v.sint);
    emFreeValue(value);
}
```

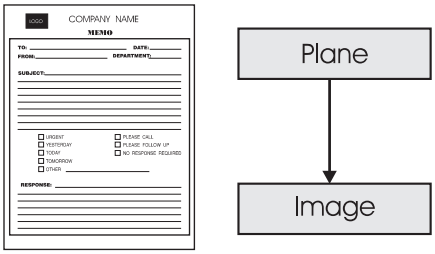
The selection list guides the e-material negotiation to the best encodings with which sender and receiver can mutually deal. The function `emMatch()` will determine if a given value is contained within a selection list. The function `emMatch()` will also check whether an integer value is part of a range:

```
if (emMatch(em, NULL, vEncoding, vImage))
{
    /* Image encoding is offered */
    ...
}
```

Surface Hierarchy Examples

The e-material encodings can be used to represent various kinds of physical material for the purpose of exchange. These examples show the document itself, the surface hierarchy diagram, and the document description table of various commonly exchanged documents. Each gray box represents a surface, whether an association, a plane, or other type, that is a building block for representing the document.

Figure 6.8 Single Page with Raster Image



A single page can be modeled using two surfaces. The top-level surface is encoded as a plane containing one child-surface encoded as an image. The image occupies most of the front side of the plane. The document description table (with many details missing) would be:

Table 6.11: Surface vDocument

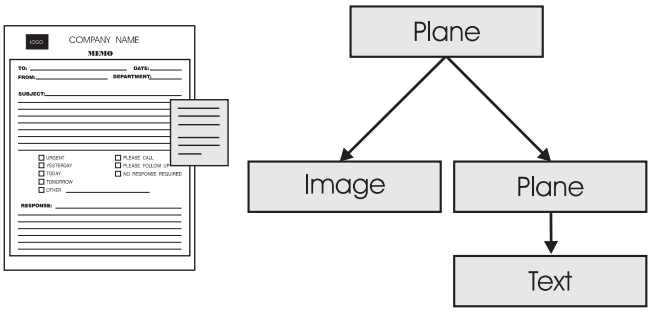
Surface	Encoding Hierarchy	Attribute	Value
Top	" "	***vEncoding***	***vPlane***
Top	***vPlane***	***vChildFront***	1 ***vSequential*** A
Top	***vPlane***	*Other Attributes*	*Other Values*

Table 6.11: Surface vDocument

Surface	Encoding Hierarchy	Attribute	Value
A	" "	***vEncoding***	***vImage***
A	***vImage***	*Other Attributes*	*Other Values*

The page could also be modeled as a single surface with an image encoding, leaving out the plane. However, the plane defines paper attributes such as size, background color, and transparency properties, which would be lost if the surface was encoded as a single surface.

Figure 6.9 Page with Sticky Note

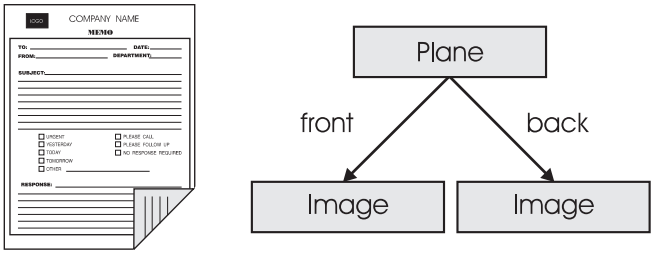


Planes can contain other planes that are like a note stuck to paper. The example shows a yellow sticky attached to a page. The sticky has text on it. The top-level plane represents the white piece of paper that has two children stuck to its front; the image and the plane represent the sticky. The second plane carries the text annotation as a child surface encoded as text. The document description table would be:

Table 6.12: Surface vDocument

Surface	Encoding Hierarchy	Attribute	Value
Top	" "	***vEncoding***	***vPlane***
Top	***vPlane***	***vChildFront***	2 ***vSequential*** A B
Top	***vPlane***	*Other Attributes*	*Other Values*
A	" "	***vEncoding***	***vImage***
A	***vImage***	*Other Attributes*	*Other Values*
B	" "	***vEncoding***	***vPlane***
B	***vPlane***	***vChildFront***	1 ***vSequential*** C
B	***vPlane***	*Other Attributes*	*Other Values*
C	" "	***vEncoding***	***vText***
C	***vText***	*Other Attributes*	*Other Values*

Figure 6.10 Double Sided Page

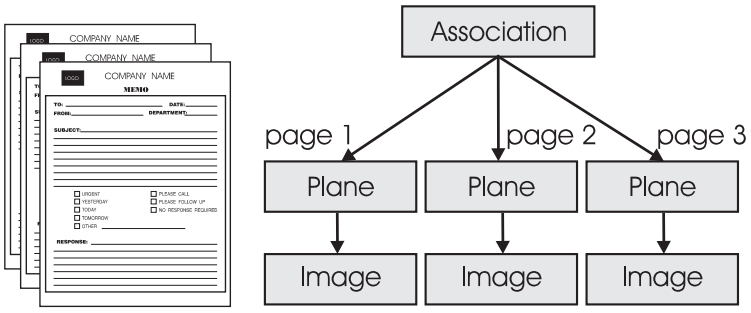


A plane has two sides, the front and the back, just like a piece of paper The previous examples showed planes with attachments on the front side only. This example shows a plane with an image on each side, modeling a double-sided piece of paper. The document description table would be:

Table 6.13: Surface vDocument

Surface	Encoding Hierarchy	Attribute	Value
Top	" "	***vEncoding***	***vPlane***
Top	***vPlane***	***vChildFront***	1 ***vSequential*** A
Top	***vPlane***	***vChildBack***	1 ***vSequential*** B
Top	***vPlane***	*Other Attributes*	*Other Values*
A	" "	***vEncoding***	***vImage***
A	***vImage***	*Other Attributes*	*Other Values*
B	" "	***vEncoding***	***vImage***
B	***vImage***	*Other Attributes*	*Other Values*

Figure 6.11 Three-Page Document



A document is an ordered sequence of pages. It is represented in e-material by an association of planes, in this case three planes, each with one image on its front side.

The document description table would be:

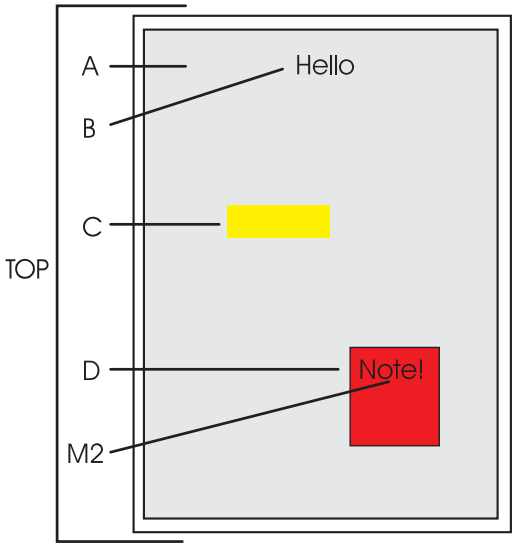
Table 6.14: Surface vDocument

Surface	Encoding Hierarchy	Attribute	Value
Top	" "	***vEncoding***	***vAssociation***
Top	***vAssociation***	***vChild***	1 ***vSequential*** A B C
A	" "	***vEncoding***	***vPlane***
A	***vPlane***	***vChildFront***	1 ***vSequential*** M1
A	***vPlane***	*Other Attributes*	*Other Values*
B	" "	***vEncoding***	***vPlane***
B	***vPlane***	***vChildFront***	1 ***vSequential*** M2
B	***vPlane***	*Other Attributes*	*Other Values*
C	" "	***vEncoding***	***vPlane***
C	***vPlane***	***vChildFront***	1 ***vSequential*** M3
C	***vPlane***	*Other Attributes*	*Other Values*
M1	" "	***vEncoding***	***vImage***
M1	***vImage***	*Other Attributes*	*Other Values*
M2	" "	***vEncoding***	***vImage***
M2	***vImage***	***Other Attributes***	*Other Values*
M3	" "	***vEncoding***	***vImage***
M3	***vImage***	*Other Attributes*	*Other Values*

Detailed Document Example

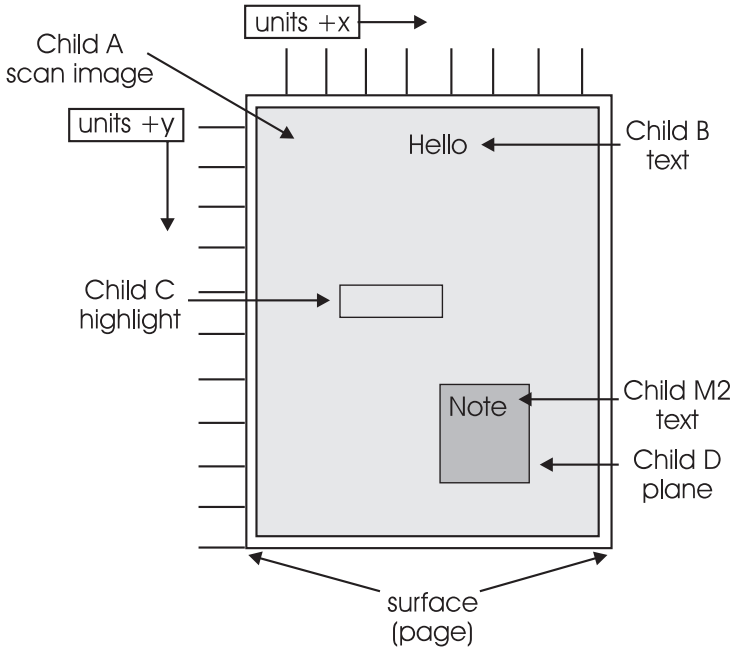
Consider the following page: It consists of a scanned page (shown in gray), with text added at the top, a yellow highlight mid-page, and a red sticky note with text. Physically, the document is commonly producable. Electronically, the document would probably be scanned with the latter three elements added after the original scan by a software application.

Figure 6.12 A Document as We Perceive It



To represent the document shown above electronically, the various elements of/on the surface must be broken down. It could be thought of as a document with four elements/ child documents. Alternatively, it could be described as only a single image, with no elements or children. E-material allows the page to be represented in a number of ways.

Figure 6.13 Document that Enumerates Structural Attributes

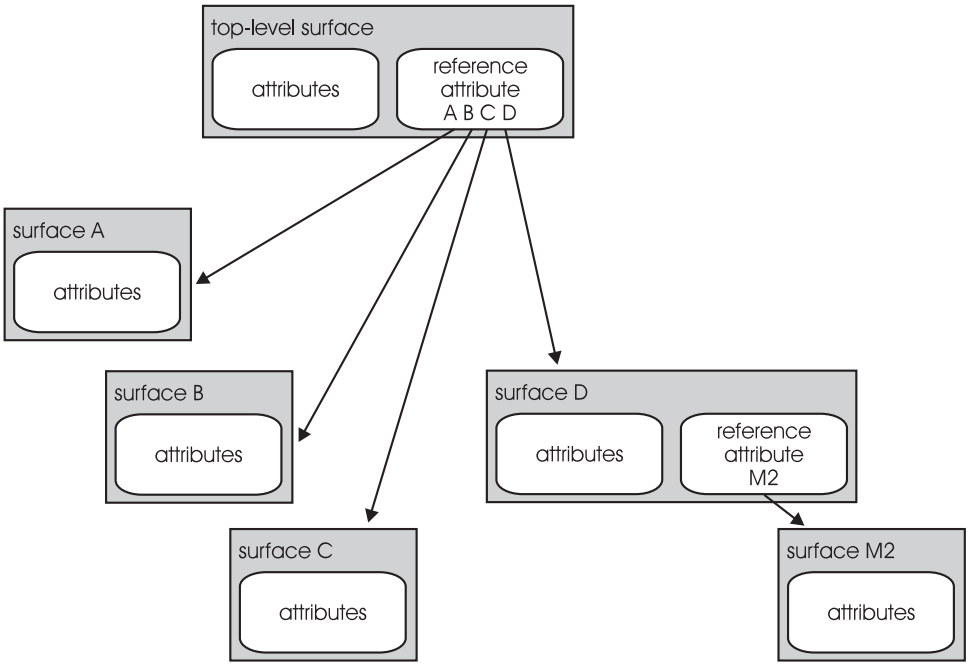


The application software that creates and edits this document must store some internal representation of the page and its structure. Hypothetically, that internal structure and organization may look something like the representation in Figure 6.13.

Consider the representation of this document in terms of the surfaces shown. The document/ op-level surface contains/links to the four child surfaces. The fourth child surface contains/links to a fifth child surface. This surface hierarchy links together six completely distinct surfaces. It is done

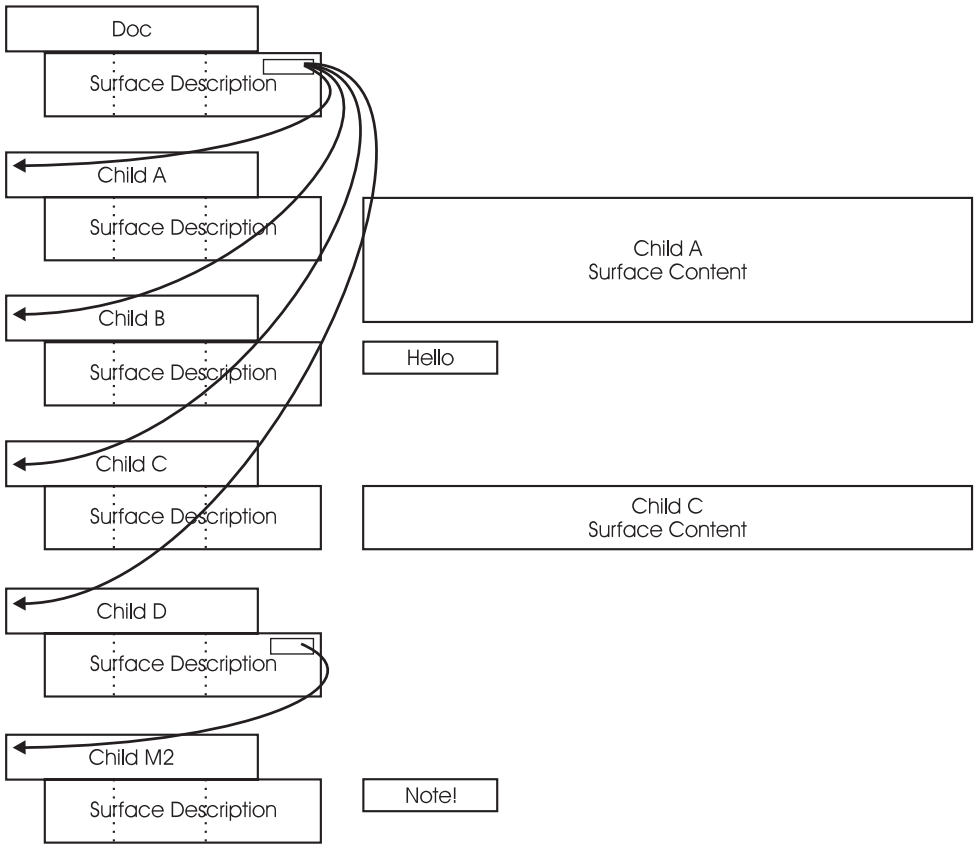
using reference lists. The child attributes are allowed to have reference lists in the value field. The surface hierarchy is shown in Figure 6.14. Except for required uniqueness, the name of each surface is independent of other surface names.

Figure 6.14 Surface Hierarchy of a Document



Each surface has a unique set of properties. These properties are found in the surface description as a set of attribute-value pairs.

Figure 6.15 Surface Hierarchy with Description Tables and Content



This same information about the surface also conveyed with a set of tables; each table expresses the properties of the corresponding surface. As seen earlier, each surface description table has three columns. The table representation contains only surface description information. The surface content information is still largely passed over.

Consider the top-level surface that describes the example document as a whole. It has been named surface Doc. It is encoded (***vEncoding*** attribute) as a page (***vPlane*** value). The front (***vChildFront*** attribute) of the page contains four additional surfaces (A, B, C, D values). The size (***vSize*** attribute) of the page is 8.5 by 11 inches (measured in 72,000ths of an inch). Note that the decision path that names the level or attribute block in which the ***vChildFront*** and ***vSize*** attributes appear is shown as ***vPlane*** in column one. This is because these attributes apply to the page (***vPlane*** level).

The attributes of the various child surfaces relating them to this parent surface are also a part of this surface description. Child A is has a position (***vPosition*** attribute) and method of attachment (***vAttachment*** attribute) that are expressed here. Note that these attributes only apply to Child A so the decision path of the attribute-value pair is ***vPlane***.A. The decision paths apply to each block of attribute-value pairs, and are written in this dot notation. Each selection decision is reflected as

a value in the decision path. The attribute-value pairs of the other child surfaces are also blocked together into a single level and given an appropriate decision path name.

Table 6.15: Surface Doc

Encoding Hierarchy	Attribute	Value
" "	***vEncoding***	***vPlane***
vPlane	***vChildFront***	4 ***vSequential*** A B C D
vPlane	***vSize***	(612000,792000)
vPlane.A	***vPosition***	(18000,18000)
vPlane.A	***vAttachment***	***vFixed***
vPlane.B	***vPosition***	(288000,72000)
vPlane.B	***vAttachment***	***vFixed***
vPlane.C	***vPosition***	(6120000,360000)
vPlane.C	***vAttachment***	***vFixed***
vPlane.D	***vPosition***	(360000,540000)
vPlane.D	***vAttachment***	***vFloating***

Each of the four child surfaces also has a surface description that can be viewed as a table. Surface A is the large image covering essentially the whole page.

Table 6.16: Child A/Surface A

Encoding Hierarchy	Attribute	Value
" "	***vEncoding***	***vImage***
vImage	***vSize***	(57600,75600)
vImage	*Other Attributes*	*Other Values*

Surface B is the text at the top of the page and uses a simple text encoding. A critical attribute of text is the language it represents. This attribute comes first in the encoding hierarchy of this simple text. Note that the language (***vLanguage*** attribute) may be in English (en attribute). English is the only choice in this encoding, but in some documents, there will be other language choices. The other choices appear as in a selection list.

Table 6.17: Child B/Surface B

Encoding Hierarchy	Attribute	Value
" "	***vLanguage***	“en”

Table 6.17: Child B/Surface B

Encoding Hierarchy	Attribute	Value
en	***vEncoding***	***vText***
en.***vText***	***vSize***	(72000,18000)
en.***vText***	***vSymbolSet***	***vAscii***
en.***vText***	***vData***	“Hello”

Like surface A, surface C is an image (***vImage*** value). The selection values for each of the various attributes (***vColorSpace***, ***vPixelDepth***, ***vResolution***, and ***vCompression***) are added to the encoding hierarchy as selections are made of their values. This expresses precedence among the attributes.

Images are characterized by a set of attributes (***vColorSpace***, ***vPixelDepth***, ***vResolution***, and ***vCompression***). Values associated with these attributes make up the decision path. The attribute vEncoding places the selection value ***vImage*** in the decision path. The attribute ***vColorSpace*** places the selection value ***vSRGB*** in the decision path. So it is with attribute ***vPixel*** and selection value 24, with attribute ***vResolution*** and selection value (150,150), attribute ***vCompression*** and selection value none.

Table 6.18: Child C /Surface C

Encoding Hierarchy	Attribute	Value
" "	***vEncoding***	***vImage***
vImage	***vSize***	(72000,18000)
vImage	***vOpacity***	100
vImage	***vColorSpace***	***vSRGB***
vImage.vSRGB	***vPixelDepth***	24
vImage.vSRGB.24	***vResolution***	(150,150)
vImage.vSRGB.24.(150,150)	***vPixels***	(150,38)
vImage.vSRGB.24.(150,150)	***vCompression***	***vNone***
vImage.vSRGB.24.(150,150).***vNone***	***vEndian***	***vMSB***

Surface D is a page (***vPlane***), but smaller than surface A. It also has a single child surface as shown by the reference list (***vChildFront*** attribute, M2 value).

The selection of surface names is left to the individual sending appliance creating the e-material description. There appears to be ample opportunity to chose names in a creative and compatible manner.

Table 6.19: Child D/Surface D

Encoding Hierarchy	Attribute	Value
" "	***vEncoding***	***vPlane***
vPlane	***vChildFront***	1 ***vSequential*** M2
vPlane	***vSize***	(72000,72000)
vPlane	***vOpacity***	255
vPlane	***vColor***	(240 20 10)
vPlane.M2	***vPosition***	(0,0)
vPlane.M2	***vAttachment***	***vFixed***

Like surface B, surface M2 is also a text surface (***vText*** attribute). Because the contents of this surface are so small, surface M2 is included in-line (***vData*** attribute).

Table 6.20: Surface M2

Encoding Hierarchy	Attribute	Value
" "	***vLanguage***	“en”
en	***vEncoding***	***vText***
en.***vText***	***vSize***	(72000,72000)
en.***vText***	***vSymbolSet***	***vAscii***
en.***vText***	***vData***	“Note!”

The table representation of surface A was left incomplete in order to allow the various properties of this representation to be explained. With an understanding of selection lists and how they are used in the encoding hierarchy, you are now ready to see the full description.

In form, surface A is similar to surface C, which is also an image surface (***vImage*** attribute). However, there are three separated encodings for these elements of the document. The sender has chosen to create three encodings for surface A: a 1-bit-per-pixel gray scale, an 8-bit-per-pixel gray scale, and a 24-bit-per-pixel color. These multiple encodings are signaled by the two attributes—***vColorSpace*** and ***vPixelDepth***—that have selections in selection lists as values.

In order for a sending appliance to offer multiple encodings for a surface, it must be able to offer the surface content in multiple ways. The sender must be able to convert the surface content in

some way. The data can be offered in two forms: in native mode and in one or more converted forms; or the data in two or more converted forms.

A receiving appliance is allowed to choose between the three different encodings of the surface offered by the sending appliance: a 1-bit-per-pixel gray scale, an 8-bit-per-pixel gray scale, and a 24-bit-per-pixel color. The 1-bit-per-pixel gray scale is a default encoding. The other two are base or optional encodings. In fact, the receiving appliance is allowed to choose none or more than one since it controls the transfer. Note that the resolution varies: 300 by 300 pixels per inch ((300,300)value) for 1-bit-per-pixel gray scale (***vImage***.***vGray***.1 encoding hierarchy) versus 150 by 150 pixels per inch ([150,150] value) for the other two encodings (***vImage***.***vGray***.8 and ***vImage***.***vSRGB***.24 encoding hierarchy).

The selection is made by the receiving appliance based on the surface description it receives for surface A. (The table summarizes that information.) It sees the two selection lists (noted by **1**) among the other attribute-value pairs and makes the selection. First it selects between gray scale (***vGray*** value) and standard red/green/blue color (***vSRGB*** value). If it chose the gray scale, then it would choose between 1 and 8 bits per pixel.

The choices made for this surface are completely independent of the choices made for other surfaces.

Table 6.21: Child A/Surface A

Encoding Hierarchy	Attribute	Value
" "	***vEncoding***	vImage
vImage	***vSize***	(576000,756000)
vImage	***vOpacity***	255
vImage	***vColorSpace***	***vGray vSRGB*** **1**
vImage.vGray	***vPixelDepth***	1 8 **1**
vImage.vGray.1	***vResolution***	(300,300)
vImage.vGray.1.(300, 300)	***vPixels***	(2400,3150)
vImage.vGray.1.(300, 300)	***vCompression***	***vNone***
vImage.vGray.1.(300, 300).***vNone***	***vEndian***	***vMSB***
vImage.vGray.8	***vResolution***	(150,150)
vImage.vGray.8.(150, 150)	***vPixels***	(1200,1575)
vImage.vGray.8.(150, 150)	***vCompression***	***vNone***

Table 6.21: Child A/Surface A

Encoding Hierarchy	Attribute	Value
vImage.vGray.8.(150, 150).***none***	***vEndian***	***vMSB***
vImage.vSRGB	***vPixelDepth***	24
vImage.vSRGB.24	***vResolution***	(150,150)
vImage.vSRGB.24.(150,150)	***vPixels***	(1200,1575)
vImage.vSRGB.24.(150,150)	***vCompression***	***vJPEG***
vImage.vSRGB.24.(150,150)***.vJPEG***	***vEndian***	***vMSB***

While the multiple table representation of a document is acceptable for most circumstances, a full document can be represented as a single four-column table. Following the lead set by the encoding hierarchy, the surface hierarchy is inserted as a column to the left; each line of the table is labeled with the element of the surface pertaining to that line.

Table 6.22: Document

Surface	Encoding Hierarchy	Attribute	Value
Top	" "	***vEncoding***	***vPlane***
Top	***vPlane***	***vChildFront***	***4 vSequential*** A B C D
Top	***vPlane***	***vSize***	(612000,792000)
Top	***vPlane.***A	***vPosition***	(18000,18000)
Top	***vPlane.***A	***vAttachment***	***vFixed***
Top	***vPlane.***B	***vPosition***	(288000,72000)
Top	***vPlane.***B	***vAttachment***	***vFixed***
Top	***vPlane.***C	***vPosition***	(612000,360000)
Top	***vPlane.***C	***vAttachment***	***vFixed***
Top	***vPlane.***D	***vPosition***	(360000,540000)
Top	***vPlane.***D	***vAttachment***	***vFloating***
A	" "	***vEncoding***	***vImage***
A	***vImage***	***vSize***	(576000,756000)
A	***vImage***	***vOpacity***	255

Table 6.22: Document

Surface	Encoding Hierarchy	Attribute	Value
A	***vImage***	***vColorSpace***	***vGray vSRGB***
A	***vImage.vGray***	***vPixelDepth***	1 8
A	***vImage.vGray.***1	***vResolution***	(300,300)
A	***vImage.vGray.***1.(300,300)	***vPixels***	(2400,3150)
A	***vImage.vGray.***1.(300,300)	***vCompression***	***vNone***
A	***vImage.vGray***.1.(300,300).***vNone***	***vEndian***	***vMSB***
A	***vImage.vGray***.8	***vResolution***	(150,150)
A	***vImage.vGray***.8.(150,150)	***vPixels***	(1200,1575)
A	***vImage.vGray***.8.(150,150)	***vCompression***	***vNone***
A	***vImage.vGray***.8.(150,150).***vNone***	***vEndian***	***vMSB***
A	***vImage.vSRGB***	***vPixelDepth***	24
A	***vImage.vSRGB***.24	***vResolution***	(150,150)
A	***vImage.vSRGB***.24.(150,150)	***vPixels***	(1200,1575)
A	***vImage.vSRGB***.24.(150,150)	***vCompression***	***vJPEG***
A	***vImage.vSRGB***.24.(150,150).***vJPEG***	***vEndian***	***vMSB***
B	" "	***vLanguage***	“en”
B	en	***vEncoding***	***vText***
B	en.***vText***	***vSize***	(7200,1800)
B	en.***vText***	***vSymbolSet***	***vAscii***
B	en.***vText***	***vData***	“Hello”
C	" "	***vEncoding***	***vImage***
C	***vImage***	***vSize***	(72000,18000)
C	***vImage***	***vOpacity***	100
C	***vImage***	***vColorSpace***	***vSRGB***
C	***vImage.vSRGB***	***vPixelDepth***	24
C	***vImage.vSRGB***.24	***vResolution***	(150,150)
C	***vImage.vSRGB***.24.(150,150)	***vPixels***	(150,38)
C	***vImage.vSRGB.***24.(150,150)	***vCompression***	***vJPEG***
C	***vImage.vSRGB***.24.(150,150).***vJPEG***	***vEndian***	***vMSB***
D	" "	***vEncoding***	***vPlane***

Table 6.22: Document

Surface	Encoding Hierarchy	Attribute	Value
D	***vPlane***	***vChildfront***	1 ***vSequential*** M2
D	***vPlane***	***vSize***	(7200,7200)
D	***vPlane***	***vOpacity***	255
D	***vPlane***	***vColor***	(240,20,10)
D	***vPlane***.M2	***vPosition***	(0,0)
D	***vPlane***.M2	***vAttachment***	***vFixed***
M2	" "	***vLanguage***	“en”
M2	en	***vEncoding***	***vText***
M2	en.***vTex***t	***vSize***	(7200,7200)
M2	en***.vText***	***vSymbolSet***	***vAscii***
M2	en.***vText***	***vData***	“Note!”

The elements in the document description table are shown here in standard order. The use of standard order is recommended. Sending appliances must adhere to this order. (Deviations from this order are not addressed here.) Fundamentally standard order is:

- Depth-first enumeration of encoding hierarchy elements.
- The first entry after an option level is used as the default in the absence of other information.

Reconsider the surface description table for Surface A. Each of the selection lists has been marked with **“1.”** Since the single element selection list containing ***vImage*** appears first (in the value column), vImage appears first in all subsequent decision paths (in the encoding hierarchy column). The next selection list contains both ***vGray*** and ***vSRGB***. As these are two mutually exclusive alternatives, one or the other appears second in each subsequent decision path. This continues throughout the surface description table. Decision paths are constructed from alternative selectable elements from the selection lists. Attributes appear which allow only single values. They do not figure in decision path construction.

Table 6.23: Surface A with Selection Lists Marked

Encoding Hierarchy	Attribute	Value
" "	***vEncoding***	***vImage*** 1
vImage	***vSize***	(576000,756000)
vImage	***vOpacity***	255
vImage	***vColorSpace***	***vGray vSRGB*** 1

Table 6.23: Surface A with Selection Lists Marked

Encoding Hierarchy	Attribute	Value
vImage.vGray	***vPixelDepth***	1 8 **1**
vImage.vGray.1	***vResolution***	(300,300) **1**
vImage.vGray.1.(300,300)	***vPixels***	(2400,3150)
vImage.vGray.1.(300,300)	***vCompression***	***vNone*** **1**
vImage.vGray.1.(300,300)***.vNone***	***vEndian***	***vMSB*** **1**
vImage.vGray.8	***vResolution***	(150,150) **1**
vImage.vGray.8.(150,150)	***vPixels***	(1200,1575)
vImage.vGray.8.(150,150)	***vCompression***	***vNone*** **1**
vImage.vGray.8.(150,150).***vNone***	***vEndian***	***vMSB*** **1**
vImage.vSRGB	***vPixelDepth***	24 **1**
vImage.vSRGB.24	***vResolution***	(150,150) **1**
vImage.vSRGB.24.(150,150)	***vPixels***	(1200,1575)
vImage.vSRGB.24.(150,150)	***vCompression***	***vJPEG*** **1**
vImage.vSRGB.24.(150,150).***vJPEG***	***vEndian***	***vMSB*** **1**

Reconsider also the document description table. The selection lists **1** and reference lists **2** each have been marked. Since attribute ***vEncoding*** appears first with the single element selection list containing ***vPlane***, ***vPlane*** appears first in all subsequent decision paths. The next list is the reference list containing surface names A, B, C, and D. Each of these then appears in decision paths.

Table 6.24: Document with Selection Lists and Reference Lists Marked

Surface	Encoding Hierarchy	Attribute	Value
Top	" "	***vEncoding***	***vPlane***
Top	***vPlane***	***vChildFront***	4 ***vSequential*** A B C D
Top	***vPlane***	***vSize***	(612000,792000)
Top	***vPlane.A***	***vPosition***	(18000,18000)
Top	***vPlane.A***	***vAttachment***	***vFixed***
Top	***vPlane.B***	***vPosition***	(288000,72000)
Top	***vPlane***.B	***vAttachment***	***vFixed***

Table 6.24: Document with Selection Lists and Reference Lists Marked

Surface	Encoding Hierarchy	Attribute	Value
Top	***vPlane***.C	***vPosition***	(6120000,360000)
Top	***vPlane.***C	***vAttachment***	***vFixed***
Top	***vPlane***.D	***vPosition***	(360000,540000)
Top	***vPlane***.D	***vAttachment***	***vFloating***
A	" "	***vEncoding***	***vImage*** **1**
A	***vImage***	***vSize***	(576000,756000)
A	***vImage***	***vOpacity***	255
A	***vImage***	***vColorSpace***	***vGray vSRGB*** **1**
A	***vImage.vGray***	***vPixelDepth***	1 8 **1**
A	***vImage.vGray***.1	***vResolution***	(300,300) **1**
A	***vImage.vGray***.1.(300,300)	***vPixels***	(2400,3150)
A	***vImage.vGray***.1.(300,300)	***vCompression***	***vNone*** **1**
A	***vImage.vGray***.1.(300,300).***vNone***	***vEndian***	***vMSB*** **1**
A	***vImage.vGray***.8	***vResolution***	(150,150) **1**
A	***vImage.vGray***.8.(150,150)	***vPixels***	(1200,1575)
A	***vImage.vGray***.8.(150,150)	***vCompression***	***vNone*** **1**
A	***vImage.vGray***.8.(150,150).***vNone***	***vEndian***	***vMSB*** **1**
A	***vImage.vSRGB***	***vPixelDepth***	24 **1**
A	***vImage.vSRGB***.24	***vResolution***	(150,150 **1**)
A	***vImage.vSRGB***.24.(150,150)	***vPixels***	(1200,1575)
A	***vImage.vSRGB***.24.(150,150)	***vCompression***	***vJPEG*** **1**
A	***vImage.vSRGB***.24.(150,150).***vJPEG***	***vEndian***	***vMSB*** **1**
B	" "	***vLanguage***	“en”**1**
B	“en”	***vEncoding***	***vText*** **1**
B	en.***vText***	***vSize***	(7200,1800)
B	en.***vText***	***vSymbolSet***	***vAscii*** **1**
B	en.***vTex***t	***vData***	“Hello”
C	“ ”	***vEncoding***	***vImage*** **1**
C	***vImage***	***vSize***	(7200,1800)
C	***vImage***	***vOpacity***	100

Table 6.24: Document with Selection Lists and Reference Lists Marked

Surface	Encoding Hierarchy	Attribute	Value
C	***vImage***	***vColorSpace***	***vSRGB*** **1**
C	***vImage.vSRGB***	***vPixelDepth***	24 **1**
C	***vImage.vSRGB***.24	***vResolution***	(150,150) **1**
C	***vImage.vSRGB***.24.(150,150)	***vPixels***	(150,38)
C	***vImage.vSRGB***.24.(150,150)	***vCompression***	***vJPEG*** **1**
C	***vImage.vSRGB***.24.(150,150).***vJPEG***	***vEndian***	***vMSB*** **1**
D	" "	***vEncoding***	***vPlane*** **1**
D	***vPlane***	***vChildfront***	1 ***vSequential*** M2 **2**
D	***vPlane***	***vSize***	(7200,7200)
D	***vPlane***	***vOpacity***	255
D	***vPlane***	***vColor***	240 20 10
D	***vPlane.***M2	***vPosition***	(0,0)
D	***vPlane***.M2	***vAttachment***	***vFixed***
M2	" "	***vLanguage***	“en” **1**
M2	“en”	***vEncoding***	***vText*** **1**
M2	en.***vText***	***vSize***	(7200,7200)
M2	en.***vText***	***vSymbolSet***	***vAscii*** **1**
M2	en.***vText***	***vData***	“Note!”

Documents have been represented as paper, documents in an electronic encoding, as diagrams representing surfaces linked together, as sets of surface description tables linked by reference lists, and as a single document description table. These are all representations— ways of thinking about surfaces, documents, and e-material.

Note, however, the only real document, at least from an JetSend perspective, is one constructed for transmission. From the outbound side, a series of library calls defines the structure of the e-material surface. The description is constructed and passed down through the device-independent API, then the surface content is placed in buffers and passed down through the API. From the inbound side, the surface description is received as an event and interpreted through a series of library calls. After the description is interpreted, the receiving device makes content requests according to its capabilities and rendering priorities. The calls to the various e-material library routines ultimately determine the real structure of the surface/document.

Chapter 7

JetSend Coding Fundamentals

Datatype EMtype

As the example below illustrates, level names attribute names, and attribute values can be of different types. They are defined in the header file by the enumeration EMtype. The current implementation of the library supports the following types:

EMtype	Description	Example of value
emKeyType	keyword (16-bit integer)	***vImage***
emIntType	32-bit signed integer	8
emIntPairType	pair of integers	(100,200)
emRangeType	range of integers	100..600
emRangePairType	pair of integer ranges	(100..600, 200..600)
emStringType	ASCII string	“fest”
emDataType	binary data	0x20 0x45 0x00 0xf2 ...
emListType	list of values	2 “fest” ***vImage*** -12
emLevelType	level (list of attribute-value pairs)	

The last three types (data, list, and level) can only be used as attribute values, not as attribute names or level names. The other types can be used in all three capacities.

Datatype EMvalue

Most of the functions that create e-material take a parameter of the EMvalue type. This is a structure that contains an EMtype field specifying which type the value is, and the data for the value that is a union of fields specific to each type. So, a variable of the type EMvalue can contain either an integer, a string, a range, a list, etc. The library defines EMvalue as follows:

```
typedef struct _EMvalue {
    EMtype    iType;
       union {
       Tu16    key;
       char*   string;
       Ts32    sint;
       EMpair  range, intPair, rangePair[2];
       struct { Tu32 iSize; Tbyte* pData; } data;
       struct { Tu16 iNumber; struct _EMvalue* pValues;} list;
    } v;
} EMvalue;
```

A device can define its own variables of the type EMvalue, insert values into it, and call functions in this library. However, the library contains some conversion functions and constants that make it easier to call other functions. The constants define keyword values and are prefixed with “v” (for “value”). There is a constant for each keyword in the e-material specification, for example, ***vEncoding***, ***vColorSpace***, ***vSize***, etc. Refer to the header file (appl_Emlib.h) for a complete list of these constants. For other values, the conversion functions can be used to convert simple types (integers, strings, etc.) to EMvalues that can be passed directly to library function calls.

Example - Creating a Description

Functions convered are emPrepare, emAdd, emAddLevel.

The following code creates the description depicted earlier in this document:

```
Tbyte*   em;
Tbyte*   level;

em = Tmalloc(2048);
emPrepare(em, 2048);

emAdd(em, NULL, vEncoding, emList(2, vImage, vText));
emAdd(em, NULL, vSize, emIntPair(100,200));

level = emAddLevel(em, NULL, vImage);
emAdd(em, level, vResolution, emRange(100, 600));
emAdd(em, level, vDepth, emInt(8));
```

```
level = emAddLevel(em, NULL, vText);
emAdd(em, level, vSymbolSet, vAscii);
emAdd(em, level, vData, emString("fest"));
```

First, a block is allocated and prepared as an e-material block (em). Second, two attributes are added to the null level with `emAdd` function calls. (Note the use of the conversion functions `emList` and `emIntPair`; this means that the caller does not have to define em value variables.) Third, the two sub-levels are added one at a time. The `emAddLevel` call returns a pointer to the level used in subsequent `emAdd` calls for attributes belonging to that level.

Example - Memory and Description

In the previous example the em block was `malloc`ed using `Tmalloc()`. Another approach is to allocate a TOPS buffer, a Tcbd or BD. So the memory allocation section in the example above would probably look as follows:

```
bd = amCreateBuf(2048);
em = BufDataAddr(bd);
emPrepare(em, BufDataSize(bd));
```

Memory for e-material is allocated above the **device independent API** and freed below that API. The `amCreateBuf`() function is used in such circumstances. Here is an example using the recommended code:

```
Tbyte*   em;
Tbyte*   level;

/* em = Tmalloc(2048);  ORIGINAL */
/* emPrepare(em, 2048);  ORIGINAL */
bd = amCreateBuf(2048);
em = BufDataAddr(bd);
emPrepare(em, BufDataSize(bd));

emAdd(em, NULL, vEncoding, emList(2, vImage, vText));
emAdd(em, NULL, vSize, emIntPair(100,200));

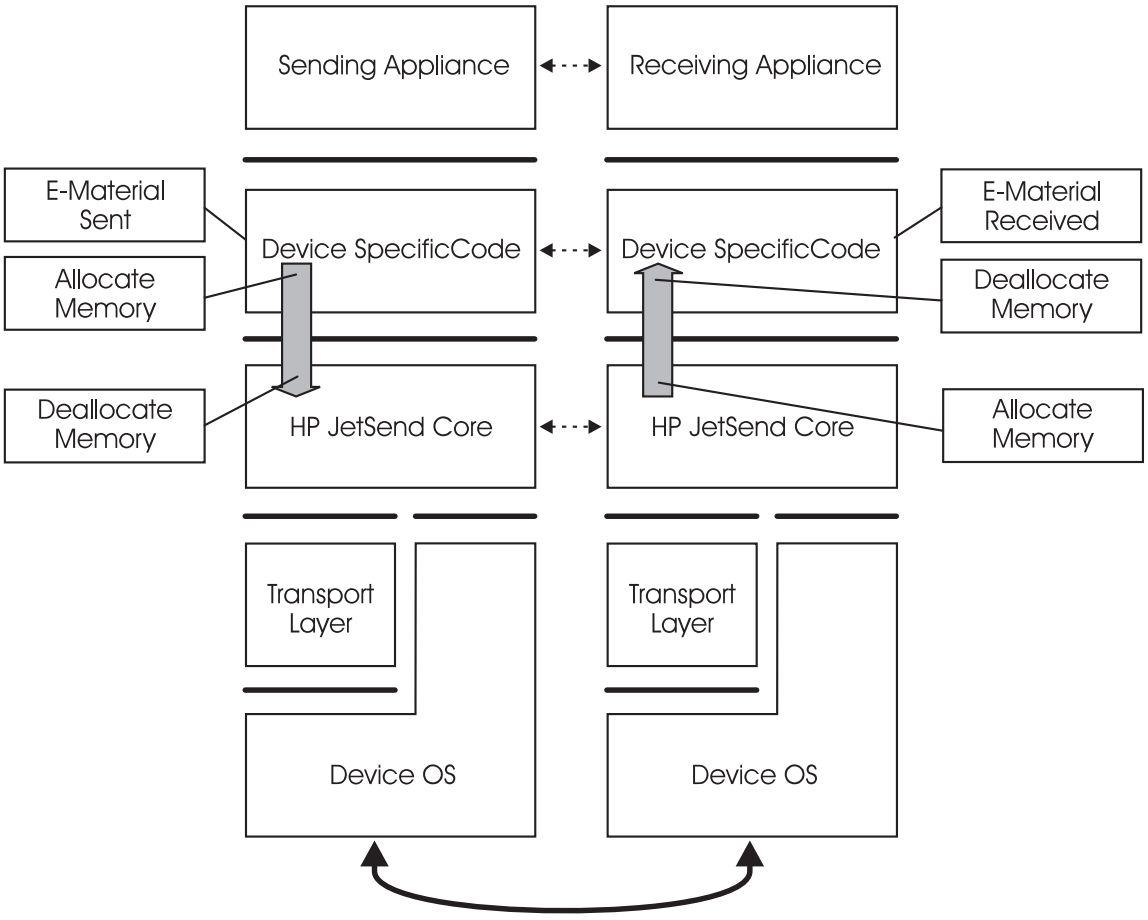
level = emAddLevel(em, NULL, vImage);
emAdd(em, level, vResolution, emRange(100, 600));
emAdd(em, level, vDepth, emInt(8));

level = emAddLevel(em, NULL, vText);
emAdd(em, level, vSymbolSet, vAscii);
emAdd(em, level, vData, emString("fest"));
```

The function `emFreeValue()` is called after the `emGet()` function. It will free the memory associated with the pointer `value.v.list.pValues` that is set if the value happens to be a list. Lists are returned as an array of EMvalues and need to be freed by the caller of `emGet()`. It is recommended that `emFreeValue()` is always called after `emGet()`, even if the caller does not expect a list value.

This point is critical to the successful implementation of a JetSend appliance. Memory for outbound e-material should be allocated using a TOPS buffer in the device-specific code. The memory will be deallocated by the JetSend core. Similarly, memory is allocated by JetSend core for inbound e-material. The device-specific code needs to deallocate the memory properly.

Figure 7.1 Allocation and Deallocation of Memory for E-Material



Example - Reading a Description

These functions are covered in the following discussion: `emMatch()`, `emGet()`, `emGetLevel()`, `emFreeValue()`. The function `emMatch()` is an easy way to check whether a particular value is part of an attribute's selection list. For example, to check whether an image encoding is offered in the e-material block em do the following:

```
if (emMatch(em, NULL, vEncoding, vImage))
{
     /* Image encoding is offered */
     ...
}
```

The function `emMatch()` will also check whether an integer value is part of a range. (See the documentation for `emMatch()` for futher details.) The function `emGet()` is used when the caller needs to know an actual value of an attribute. The function `emGet()` returns an EMvalue variable. The following example illustrates getting the image sub-level and retrieving the value of the depth attribute using `emGet()`:

```
EMvalue value;

level = emGetLevel(em, NULL, vImage);
if (level && emGet(em, level, vDepth, &value))
{
     if (value.iType == emIntType)
        printf("The depth is %d\n", value.v.sint);

     emFreeValue(value);
}
```

Restrictions

The library has a few restrictions that exist only for implementation reasons and may be removed in future revisions of the library. The most important restriction follows:

Attributes and sublevels cannot be added to a level that has already been closed. A level is closed when attributes or sublevels have been added to the parent of the level after the level was created. This means `emAddLevel()` and `emAdd()` must be called in the right order when building up a description. Here is an illegal sequence:

```
1. level = emAddLevel(em, NULL, vImage);
2. emAdd(em, level, vStuff, vStuffed);
3. emAdd(em, NULL, vMoreStuff, vReallyStuffed);
4. emAdd(em, level, vWibble, vWobble);
```

The level was closed in line 3 because an attribute was added to the parent (null). Line 4 will fail because the library cannot insert attributes into closed levels.

The function `emAdd(...,Level,...)` does not check if an attribute exists already when it is added. The functions `emGet()` and `emFind()` will return the first one added. A future revision of this library might change the value of an existing attribute. It is therefore recommended that no attributes are added using the same name.

JetSend APIs

Transport Independent API (TIAPI)

TIAPI is the interface by which JetSend core hooks into the network functionality of a device. TIAPI can support multiple transport layers such as TCP/IP, IrDA, loop-back, and so forth. The developer of any given device decides which transports are made available to JetSend core through TIAPI. The developer needs to write code to map the TIAPI calls to network calls, such as Berkeley Sockets, on a given platform. This code is referred to as the network "glue."

The ADK includes network glue for TCP/IP mapped to Berkeley Sockets as well as a loop-back transport layer that allows a device to talk to itself . On devices not interfacing the network through Berkeley Sockets or using another transport, the network glue will have to be written by the device developers.

System API

The JetSend core code uses functions to allocate memory and functions for using timers. This interface to the device's operating system is defined by the system API. In the code provided with the ADK, all the system functions are mapped to standard C library functions (stdlib). Developers can map these function calls differently if it is required on their platform.

Device API

This section contains an overview of the functionality that the JetSend Communications Technology provides to a device.[1] The device API is split a number of modules as depicted below.

Device Independent API

Device independent API is called by the device code and provides the functionality of JetSend core. It includes functions for sending and receiving information, providing and monitoring status messages, and more. Events are passed from JetSend core to the device by polling. The following overall functionality is available through device independent API:

- Connect using a standard bi-directional transport layer, such as TCP/IP, infrared, or others. Exchange names, icons and other general information.

1. Future revisions of JetSend will be enhanced with more functionality. This includes device and job control, as well as security features. Future revisions will be backwards-compatible.

- Send and receive jobs. A job is typically a document with pages that one device is sending to the other. The actual document encoding is negotiated using e-material.
- Exchange status information. Devices can exchange both device and job status. Device status reflects the general state of a device, whereas job status pertains to a specific job being processed.
- Exchange device addresses. A device can send destination addresses to another device. This is useful for programming devices with simple user interfaces from devices with more complex ones.
- Security mechanisms for IP filtering and password exchange.

Events

Functions: amGetNextEvent, amFreeEvent

Each appliance receives information through events. An event is received through the network and enqueued by JetSend core. This queue is then polled using `amGetNextEvent()` to process the events. Event arrive is independent of every processing, since the polling is asynchronous (non-blocking).

Events may be quite simple, or of the event-allocated memory is eased using `amFreeEvent()`.

Session Functions

Functions: amStartSession, amEndSession, amListenForSession
Events: amSessionStartEvent, amSessionEndEvent

In order to interact, two JetSend-enabled devices have to establish a session. JetSend core provides functions and events to start and end a session with remote devices. To view the status of another device, send jobs or even exchange names, a session needs to be established first.

A device can issue requests to start sessions with another device if it knows its address (`amStartSession()`). Both ends will receive events to notify when a session has been established (`amSessionStartEvent()`). They will also be notified when the session terminates, which happens if either one of the devices ends the session explicitly (`amEndSession()`) or if a time out occurs in the network layers.

On networks with auto-discovery, such as infrared (IrDA), two devices will be notified of new sessions automatically.

New sessions are identified with a session handle that is valid for the duration of a session. This handle is used to refer to the session in various function calls and events in other parts of the API.

Each session takes up a small amount of memory in JetSend core. Devices that have a limited amount of RAM can either end incoming sessions or disallow new sessions (amListenForSession) as required.

Contact Module

Functions: con*

Events: con*

When devices start a session, the contact activity on each end will exchange self surfaces. This is a structure with general information about a device; the following related components are defined:

- Label (ASCII string)—Text string that describes the device (“Tom’s Printer,” or “Lynn’s PC,” for example).
- Icon (image)—Small image that represents the device. A device can change its image as a way of reflecting status, although most devices will probably keep their images static.

Devices choose which of the components they provide (`conSetComponent()`) and which they receive from others (`conInitialize()`). None of the components is required, but it is highly recommended to provide at least the label.

Job Modules (Sender and Receiver)

Functions: ps*, pr*

Events: ps*, pr*

The major task of information transfer is completed by the job module. The sending appliance has previously opened a session with the receiving appliance. It has the surface description and surface content of one or more surfaces at its disposal. One of the appliances (sender or receiver) initiates a transfer of that information. The sender may initiate the transfer using a `prSendSurface()` call with a previously constructed surface description. This results in a `prJobStartEvent()` event at the receiver.

The e-material description is received and interpreted by the receiver using the e-material libraries. If the receiver determines that the description is incomplete, it constructs a surface description request using the attribute names and attribute values in the partial surface descrition. These are sent to the sender using a `prRequestDescription()` call, shich results in a `prDescriptionRequestEvent()` event at the sender. To fulfill this request, the sender constructs a description of the subtree of the encoding hierarchy again ustin the e-material libraries. This is transferred using `psSendDescription()` to the receiver and arrives as a `psDescriptionEvent` event. Depending on the size of the e-material description, this process can be repeated anumber of times until the entire description is complete.

The receiving appliance selects one attribute value from each selection list when formulating a surface description request. This is the negotiation step fundamental to JetSend. When the entire description has been received, the receiver formulates a content request in a manner similar to the description request. The attribute name and an attribute value from each selection list is chosen. These are used to construct a content request.`prRequestContent()` call. The sender receives this as a `psContentRequestEvent()`. The sender is obliged to fill this request by transforming the content from its native form into the form described in the content request. The content, in proper form, is transferred using `psSendContent()` and arrives at the receiver as a `prContentEvent`. The receiver interprets this raw data as required. As with the description, the content may require several interactions to transfer completely.

The bulk of information handled by JetSend will be handled by the job module.

Addressing

When an appliance initiates a session, it contacts the remote JetSend-enabled appliance using a network reference such as a URI. This is the general syntax of a JetSend address:

```
<network type>:<network address>[#<sub-address>]
```

An appliance can have one or more networks installed, so the address includes information about which network type the address pertains to. Gateways also manifest themselves as network types. The number of network types an appliance can use is therefore the number of different transports in the appliance, plus the number of gateways that the appliance can "see."

The network address is an ASCII string. The syntax is determined by the network type. The sub-address is optional. It is used to distinguish multiple logical appliances with the same physical network address, such as multiple software components on a PC.

Here are some typical examples of JetSend addresses:

ip:15.99.99.254	Simple IP address (network type "ip")
ip:15.99.99.254	IP address with sub-address (#2)
ip:smith.company	IP address using hostname (requires HNR in device)
fax:1-208-555-1212	Address of fax machine (using a fax gateway, network "fax")
http://www.hp.com	Web-address (using a web gateway, network type "http")

Address formats should be considered when designing control panels on sending devices.

IDs

The function calls and events employ a number of different IDs and handles. It is important to distinguish their purpose:

Session IDs

Each time a new session is opened, a unique session ID is allocated. It is used in subsequent calls to identify a specific session. Once a session is closed, the session ID becomes invalid to core and might be reused when a new session is opened. (This implementation increments a 16-bit number each time a session ID is allocated, so IDs are not likely to be reused for quite a while. Session IDs are allocated by the JSP layer and communicated in JSP protocol messages.

Surface Handle

A surface handle is an identifier for a surface, both expressions and impressions; it should not be confused with the surface name, which is an identifier used for requesting new surface impressions. Each time a surface is created, an ID is allocated for it. When the surface is impressed on another device (using `psSendSurface`, for example), the remote device will create a new impression for that surface and allocate a new ID. This ID is not the same as the ID created on the original device. However, both IDs will be locally unique.

The surface ID is used to identify the surface when performing operations on it, for example, when requesting its contents, updating it, or deleting it. The surface IDs live for as long as the surface exist. Expressions live until they are deleted, or until a device is turned off. Impressions live until they are unimpressed, their corresponding expression is deleted, or the session they relate to is closed.

When a surface is requested, the surface's name is used. No handle is allocated at this point. This will happen when a new surface is created in response to the surface request.

Request ID

Request IDs are temporary identifiers that are used to link a request to its response. They are necessary because requests and responses are asynchronous and may be made in parallel. There are many types of requests: surface requests, description requests, etc. For each request, a unique ID is allocated by the requesting device. The responding device will use this when it sends the response, which the requestor recognizes when it sees it.

Request IDs live until the request had been satisfied, and then they become invalid and may be reused.

Job ID

For each job on the sending and receiving side, a unique job ID is allocated. It is very similar to a surface handle in terms of allocation and lifespan. It is allocated when the job is first initiated. A different job ID is allocated on the sending and receiving side, but is unique to each side. The job ID becomes invalid once the job is terminated. Job IDs are used in the job activity (push) and to identify job status strings.

Status Modules (Provider and Observer)

Function: ss*, sr*

Events: ss*, sr*

Devices use the status module to send and receive status information at both a device level and a job level. It is not compulsory for devices to supply any status, but it is recommended to supply a minimum level of device status.

A device can have just one device status, which should reflect the general state of the device. Examples include: “toner low”, “scanning”, “ready”, and “broken”. Furthermore, a device can create a job status for each job it is processing. Examples include “printing page 4 of 12”, “in queue”, “failed” and “toast is ready”.

Whether or not devices provide job status will depend on the type of device. Sending devices do not need to provide job status—they are the consumers of job status from receiving devices. It is recommended that receiving devices provide at least a final job status for each job they handle, allowing the sender to determine whether the job has been successfully processed. The job status should not just reflect the transfer of material to the receiving device, but also any subsequent progress of the job (the printing or faxing of a document, for example).

The question of how long job status should be retained after the e-material has been transferred is a design decision of the device. Receiving devices can delete the job status for any completed job immediately, or choose to keep a record of jobs for some fixed period of time. Similarly, sending devices can keep a record of final job status for any jobs they initiated for a certain period of time. While a job is in progress, the sending device can choose to receive changes to the job status surface in order to track the job progress.

Status Values

Both device status and job status have two components, a machine interpretable status value and an e-material string describing the status. The status value contains device-independent values that basically allow a device to determine whether something has failed or not. Device-specific status messages, such as a message explaining why something failed, are put in the status string.

Below are the generic status values for device status and job status. A device is not allowed to extend these with its own values.

- `STAT_DEV_READY` - The device is ready to perform its primary function.
- `STAT_DEV_BUSY` - The device is currently performing its primary function.
- `STAT_DEV_OPERATOR` - The device requires an operator’s assistance.
- `STAT_DEV_BROKEN` - The device is not able to perform its primary function.
- `STAT_JOB_OK` - The job has been successfully processed.
- `STAT_JOB_FAIL` - The job has been canceled or something failed when processing it.

- `STAT_JOB_PENDING` - The job has been put on hold and will be processed later.
- `STAT_JOB_PROGRESS` - The job is being processed.
- `STAT_JOB_PERCENTAGE` - The job is being processed (includes a percentage to indicate how far the processing has progressed).

The device status explanations are a bit cryptic because they cover the status of both senders and receivers. For a receiver, READY means it is ready to accept new jobs, BUSY means it will accept new jobs but won't process them immediately, and BROKEN means it cannot process new jobs. For a sender, READY means it is ready to send a new job, BUSY means that it is currently sending a job, and BROKEN means it is not capable of sending jobs at the moment.

The textual status is an e-material text surface which, unlike the status value, is intended to be displayed to a user directly.

Localization

Status messages can be localized. The status provider can make each message available in different languages, and the observer can select the preferred language. Any string sent across by any activity as e-material can be localized using this mechanism, but in this API localization is limited to the strings sent by the status provider.

The status provider gives the infrastructure a list of languages for which it can provide status. One of those languages is the default language, which will be sent in-line with the e-material description of each status message.

The status observer tells the infrastructure in which language it prefers to receive messages. Depending on the remote device, that language may or may not be available. If it is unavailable, the default language of the provider will be picked.

E-material allows senders to provide a different language selection for each string and for receivers to base their selection on more complex criteria than just matching to one language. For simplicity, however, this API only allows a fixed set of languages on the sender side and a fixed preferred language on the receiver side.

Address Modules (Sender and Receiver)

Functions: as*, ar*

Events: as*, ar*

This activity enables appliances to pass the addresses of appliances they know about to a connected appliance. The activity uses the well-known address surface to "push" addresses from one appliance to another. This can be used to program devices that don't have a user interface designed for address entry. Also, if a user has a list of addresses on one sending device, one copy of the list can be dumped onto another.

An address is passed across as an address card. An address card has the same format as a self surface generated by the contact activity. The functions used to send and receive address cards use

a structure, `ADDRESS`, to send and receive the information. Some of the fields of this structure can be `NULL` but the address should always be set. Following are the fields in an address card:

- Address (ASCII string)—the address of the remote device
- Label (ASCII string)—the name of the remote device
- Icon (image)—the icon of the remote device

Note that all components can potentially be out of date when received by an address receiver. For example, if an appliance changes its name, it obviously won't be reflected in address cards with its address on it. Receivers of address cards can get up-to-date components by establishing a session with the device itself.

Typically, only sending devices are interested in addresses, since they are the ones initiating sessions. Devices that only receive, such as printers, will not implement the address activity.

Chapter 8

Transport Independent API Reference Implementation

The transport-independent API (TIAPI) provides the JetSend core with a common API to the transport service providers within the device. TIAPI can be thought of as a layer of glue that attaches the JetSend core to the device specific, network-connectivity implementation.

The TIAPI is a service, not a protocol. It simply provides a single common API to multiple transports. The protocols used under the TIAPI are the ones defined by the transports that are installed and active on the host platform. The transport is selected by the JetSend core when it makes a TIAPI function call.

JetSend core could have chosen an industry-standard transport API such as Berkeley Sockets or TLI as the TIAPI. But neither of those fits into the JetSend architecture. JetSend architecture requires that all TIAPI calls be asynchronous. Also, the architecture defines a specific method for passing information from TIAPI to JetSend core that neither Berkeley Sockets or TLI provides. TIAPI is simpler than Berkeley Sockets and TLI. This allows the JetSend Protocol Stack code to be free of transport API complexities that may obscure the JetSend core's own algorithms. However, for clarity and simplicity, TIAPI will be discussed in terms of Berkeley Sockets. The discussion of TIAPI in terms of Berkeley Sockets will not obscure the central algorithm with error and special case handling; its purpose is to further illuminate the functionality provided, not to be an exhaustive detailed internal design or example implementation of the functions.

The API consists of all the constant, type, function, and event definitions that the JetSend core needs to access the transport service providers within the local host. [1] The TIAPI contains functions and replies. Functions pass instructions and data into the TIAPI. Replies pass data the other way, from TIAPI to JetSend core. Any function that may send or receive data over the network is implemented as an asynchronous (non-blocking) call. The results of its action are returned as a corresponding reply.

The JetSend session protocol resides in the JSP, a part of the JetSend core. That is the entity that manages JetSend session protocol issues like routing to JetSend core modules based on JSP port number. It is the user of the TIAPI.

TIAPI uses three well-known ports for each network domain. One is the datagram transport port over which RMTP will run (in the INET domain this is a UDP port). The second is the reliable connection oriented transport port (in the INET domain this is a TCP port). The third is the

1. Events are referred to as replies in this API.

datagram port through which JSP gateway messages are sent and received (in the INET domain this is a UDP port).

For each domain, the well-known ports are registered with the appropriate authority. In the case of the INET domain, the ports are registered through the Internet Assigned Numbers Authority (IANA). For the SIPX domain, the numbers are registered through Novell; for the INET domain the UDP port number is 1782, the TCP port is 1782, and the gateway port is 1803.

When TIAPI announces the availability of a local network address to JSP (via the `TrplyNwkStat` reply), it fills in the port field of the network address with the datagram transport port over which RMTP will run. This is the JetSend contact address that should be displayed to the user of the local hosts as the address of their JetSend device.

For multiport boxes to provide pass-through support for JetSend, a unique set of the three well-known transport ports is needed for each of the parallel ports on the box. TIAPI is responsible for finding out which, if any, parallel port its host is attached to, and using the correct set of well-known transport ports for it.

TIAPI supports two types of communications services: message and stream. Message service uses the Reliable Message Transport Protocol (RMTP), running over the datagram protocol that is available in the domain such as UDP, IPX, etc. Stream service uses the standard connection-mode, full-duplex, reliable transports such as TCP, SPX, etc.

Core uses a `TconnHdl` to identify a connection. TIAPI creates the `TconnHdl` when a connection is established and passes it up to the JetSend core. From that point on, the core uses the `TconnHdl` to identify to the TIAPI the connection upon which the TIAPI should perform an operation. It is unimportant that the JetSend core know what the `TconnHdl` is; it should not de-reference it as if it were a pointer and the actual value is meaningless to core.

Any function for which there is an asynchronous reply includes a request handle parameter. The handle can be used by core to match-up replies with their corresponding requests. The handle is a value set by core before it calls the request function. The same handle value is returned to core in the reply that corresponds to that request call. Some requests, such as `xptListen()`, may generate multiple replies, one for each connection that results from the request. In those cases, the request handle from the original request call is returned in each reply. Similar to the `TconnHdl` for JetSend core, it is unimportant that TIAPI know what the request handle is. It only passes the request handle through from the function call into TIAPI to the corresponding resulting reply.

TIAPI provides an unsolicited reply, `TrplyNwkStat`, that tells JetSend core that a local network address is active or that the local network address has changed to inactive. These replies are not the direct result of a prior request, and will have a special request handle value of -1. (See the description section of `TrplyNwkStat` to find out what is meant by local addresses becoming active or inactive.)

A connection may be either shutdown or aborted. If it is shutdown , the underlying transport is asked to close the connection if it has that capability. It does not mean that TIAPI will complete all `xptSend()` requests before closing the connection. Any pending `xptSend()` requests being worked on by TIAPI when `xptHangUp()` is called will be cancelled.

Chapter 9

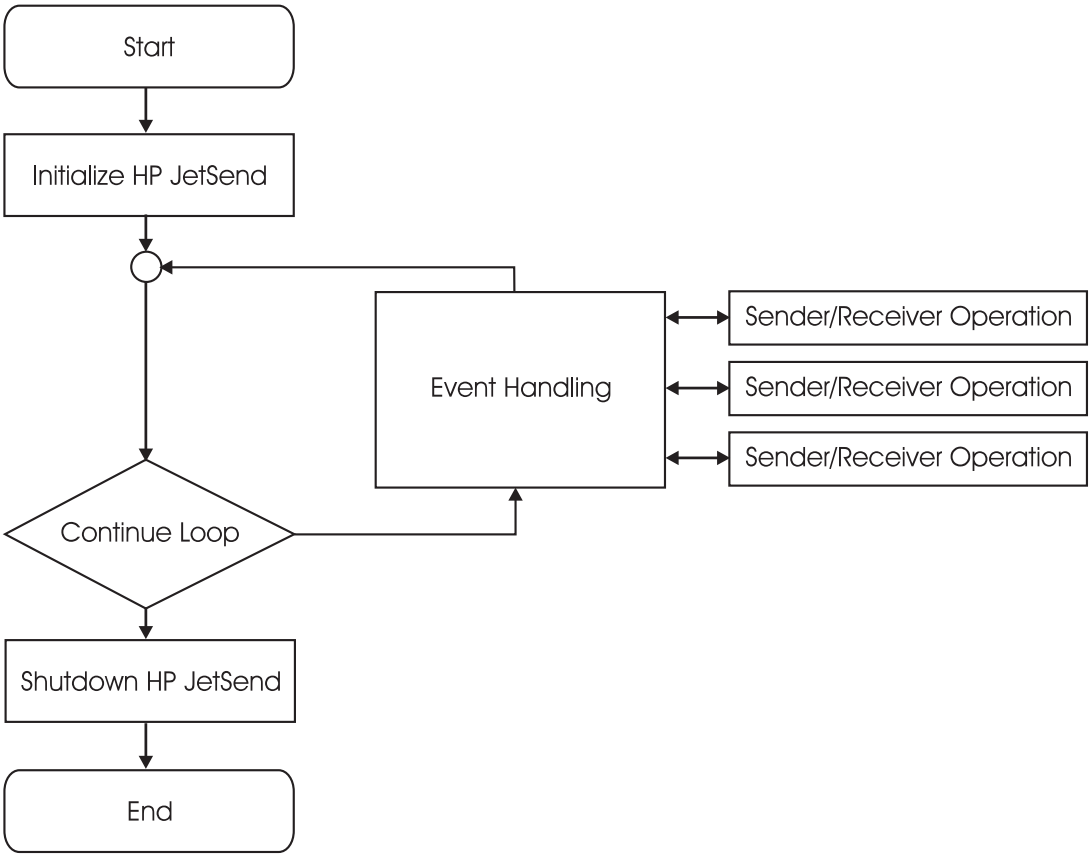
HP JetSend Sample Code

Overview

The JetSend ADK includes sample code that sends and receives various types of e-material. For each sample, there is a sending program and a receiving program that can be compiled and run on most development systems, such as a PC or UNIX workstation; an ANSI C compiler and UDP/IP network connection are necessary. Each sample pair addresses one type of encoding: image, text, file, page, or association. They are presented here in order of increasing complexity.

JetSend designates one peer appliance in a transaction as the sender, the other as the receiver. In spite of this clear distinction, the code which underlies these two distinct functionalities is structured very nearly the same for both the sending and receiving appliance. shows the general control flow for an appliance sample.

Figure 9.1 Control Flow in Sample Code



The main function of the software is to set up a loop in which events can be handled. This is true for both the sending and receiving appliances. The events that are handled differ between the sender and receiver, with the following sets of distinct functions defined in the device independent API :

- push send (ps*) versus push receive (pr*)
- status send (ss*) versus status receive (sr*)
- address send (as*) versus address receive (ar*)

Nevertheless, with the exception of the specific operations shown, the structure of the two appliances are very similar.

The sample code reflects the operations of an appliance that has an event handling loop with a large case statement for handling the various events. The various cases implement a state transition map as the appliance moves from state to state in response to various events. Each event causes the appliance to reply with an operation.

For the sending appliance, there are four events that cause the sending appliance to do significant work. (The details are in the sample code itself.) The four operations are:

- Initialize a session. Get the IP address of the receiving appliance. Open up a connection to the receiving appliance in preparation for sending a surface description.
- Specify the e-material to be sent. This can be a simple as getting a string to encapsulate into an e-material description for sending, or it may be more complex.
- Construct the e-material to be sent, based on the description. In these samples various stock files are prepared for each possible encoding. In real appliances, manipulation of the native form of the information to the correct encoding is anticipated.
- Handle any child surfaces. The full surface structure of the document must be offered.

Similarly for the receiving appliance, the sample handles several noteworthy operations:

- Interpret surface description. On receipt of the description, determine which selections are to be taken from the various selection lists.
- Request content. From the values taken from the selection lists, content requests are formulated and sent.
- Handle incoming content. The content replies must be processed.
- Render the surface. Once content is received, the receiving appliance must do something with the information. In the case of the samples, it is simply stored. In real appliances, more extensive operations would be expected.
- Handle any child surfaces. Starting with the top-level surface, each surface must be addressed.

In studying the sample code, whether sender or receiver, the basic structure remains the same. Most of the changes from sample to sample will expand on the various operations listed above.

Encodings

The current JetSend static image e-material definition system supports five top-level encodings. They are text, file image, plane, and association. A detailed explanation of these encodings is provided in the "JetSend E-Material Specification" on page A-1 of the ADK development guide. Text, file, and image are the fundamental encodings and have formats of data that can be rendered on a page. Plane and association encodings are representations of a combination of these fundamental encodings. Plane encoding can also have sub-surfaces of associations and planes, and association encoding can also have sub-surfaces of planes and associations. Plane and association encodings will be referred to in this document as **higher level encodings**.

A **plane** can be thought of as a page with a front and a back. Further, each side can be composed of images and text, perhaps in combination with associations or other planes.

An **association** simply groups together other e-materials, whether associations, planes, or the fundamental encodings.

While all encodings are valid under an association top-level encoding, file encoding is invalid under plane encoding. This is because a file cannot be rendered as such, although the text of a file, if encoded as text or image, can be rendered on a plane.

Samples

The sample code is presented as six operating pairs. These six pairs of programs are listed below with brief descriptions. A detailed description of how each program works will be presented later.

1. **In-line Text**—Sends text in-line with the e-material description to the receiver.

   The sender transmits the user's text to the receiver. The text is sent in-line with the e-material description, so there is no negotiation between the sender and receiver for the format of the text to be sent; all the data and the e-material description are sent together in a single exchange.

2. **Text**—Sends text to the receiver after the e-material description has been sent to the receiver and a `psContentRequestEvent` message has been received.

   The sender first sends a text encoding e-material description to the receiver. After the receiver makes its preferred selections and requests the content (text data), it results in a `psContentRequestEvent` message being sent to the sender. After this event is received, the sender sends to the receiver content based on the receiver's selections.

3. **File**—Sends any file specified by the user to the receiver.

   The sender first sends a file encoding e-material description to the receiver. After the receiver makes its preferred selections and requests the content (file data), it results in a `psContentRequestEvent` message being sent to the sender. After this event is received, the sender sends the data to the receiver based on the receiver's selections.

4. **Image**—Sends a Windows BMP bitmap image to the receiver.

   The sender first sends an image encoding e-material description to the receiver. After the receiver makes its preferred selections and requests the content (image data) , the sender

receives a `psContentRequestEvent` message. After this event is received, the sender sends the data to the receiver based on the receiver's selections.

5. **Plane**—Sends a plane with embedded image and text to the receiver.

   The sender first sends a top-level description to the receiver. When the receiver requests the first child's description (`psContentRequestEvent` message), the sender sends that child's description. The receiver then requests the surface description of the same child. For this request, the sender receives a psSurfaceRequestEvent message. The sender then sends the child's encoding e-material description. After the receiver receives this surface description, it makes its preferred selections and requests the content (data) of this child (`psContentRequestEvent` message). The sender then sends the data to the receiver, based on the receiver's selected preferences. This is done for each child, text, and image.

6. **Association**—Sends an association of file and image to the receiver.

   The sender first sends each child's encoding e-material description to the receiver. After the receiver makes the appropriate selections and requests the content (child data), the sender receives a `psContentRequestEvent` message. After this event is received, the sender sends the data in the requested format to the receiver. This is done for each child, file and image.

Notes on Sender Sample Programs

The sender programs are designed to be linear in the sense that they send only one job and at the end of that job, i.e., after receiving a `psJobEndEvent` message, they end the session and the program, in that order. They also establish only one session for each run.

The programs are intended to show how to send data for each of the JetSend supported encodings. To that end, there are some hard coded constants so that the methodology for achieving this objective can be seen more clearly.

There is a considerable amount of error checking, which, in some cases, shows how to verify the validity of some of the JetSend constructs in data received from the receiver. In other cases, there is very little, since it would add little to the example but clutter.

All the sender programs in this set of modules, especially those involving the higher-level encodings, process children sequentially. This means that each child is processed completely, that the data for the current child is sent to the receiver before any processing of the next child is allowed. This **sequential** attribute is represented in the JetSend system and specified in the E-material Specifications by the key vSequential.

All the sample programs follow the same control structure shown in the diagram below.

Control Flow of the Sender Sample Programs

The JetSend system must be initialized before any processing can occur successfully. This is done by calling the function `InitHPJetSend()`. This function calls `ospSetup()` first and then `amInitialize()`. The function `amInitialize()` must be called with the appropriate subaddress. The function `ospSetup()` initializes the JSP layer; this must be done before any other function calls are made. After this, `amInitialize()` must be called to initialize the activity manager (am). If this call is successful, the different modules to be used must be initialized separately. For instance, the sender must initialize the push-sender (ps) module by calling `psInitialize()`. The order in which these modules are initialized is irrelevant.

After the system has been initialized successfully, the program enters an infinite loop in which it calls the function `GetAndProcessEvent()`. In this function, events are retrieved, printed out, and processed. An event is retrieved by calling `amGetNextEvent()`. If the event contains valid data it is processed but NULL events are ignored. A NULL event means that there is no event in the system queue. If the event contains valid data, it is processed, and the program stays in the loop until it terminates. Virtually all the processing is done in this function.

When the program enters the `GetAndProcessEvent()` function, `amNetworkEvent` message must be received before any functions are called. After `amNetworkEvent` message is received, `amStartSession()` must be called to establish a session before other functions are called. It is implicit in the design of the JetSend system that senders initiate sessions. In contrast, either sender or receiver may initiate a job.

When a session is established successfully, the sender receives an `amSessionStartEvent` message with the status of AM_ACTIVE_OK. The receiver to which the sender is connected receives the same event but with a corresponding `AM_PASSIVE_OK` status. Subsequently, the sender must receive a `psInArrivedEvent` message before any e-material transmissions

between the sender and receiver can occur successfully. The `psInArrivedEvent` message event is sent automatically by the system to signify that e-material exchanges, for descriptions and data, can proceed successfully.

After `psInArrivedEvent` message has been received, the sender can send a job to the receiver—the *real* communications can begin after the `psInArrivedEvent` message has been received. A job is sent to the receiver by sending the job's encoding description in a `psStartJob()` call.

If the content data is not in-line, the sender will get a `psContentRequestEvent` message. If the encoding is fundamental, the content request will be for the data. Alternatively, if the encoding is higher level, such as plane or association, the content request could be for the description of a child, one of the components under the higher-level encoding, or the request may be for the content data of a child. If the request is for the description of a child, then the child's surface description will be requested after the receiver has received the description of the child. The request for the child's surface description comes in a `psSurfaceRequestEvent` message. Finally, the child's content data will be requested after the receiver has received the child's surface description. The data sent to the receiver will be based on the selections made by the receiver in the content request. This process is duplicated for all the children. Even if the content data is sent in-line, the receiver may request the content data via a `psContentRequestEvent` message.

The steps to send data to a receiver are summarized as follows:

- Call `ospSetup()`. This must be done to set up the JSP layer properly, even before any initialization routines such as `amInitialize()` are called.
- Initialize the JetSend system.
- Wait for the `amNetworkEvent` message.
- Call `amStartSession()`
- Wait for the `amSessionStartEvent` message
- Wait for the `psInArrivedEvent` message
- Build and send the e-material description for the encoding to be used. If the data is to be sent in-line, include it with the description.
- Wait for a `psContentRequestEvent` message if the data was not in-line, that is, if the data was not included with the e-material description. Even if the content data is included with the surface description, the receiver may request the data after it has received the description with the attached data.
- Send the content data to the receiver. The content may be the actual child data, if the encoding is fundamental, or it may be the description of a child, if the encoding higher-level.

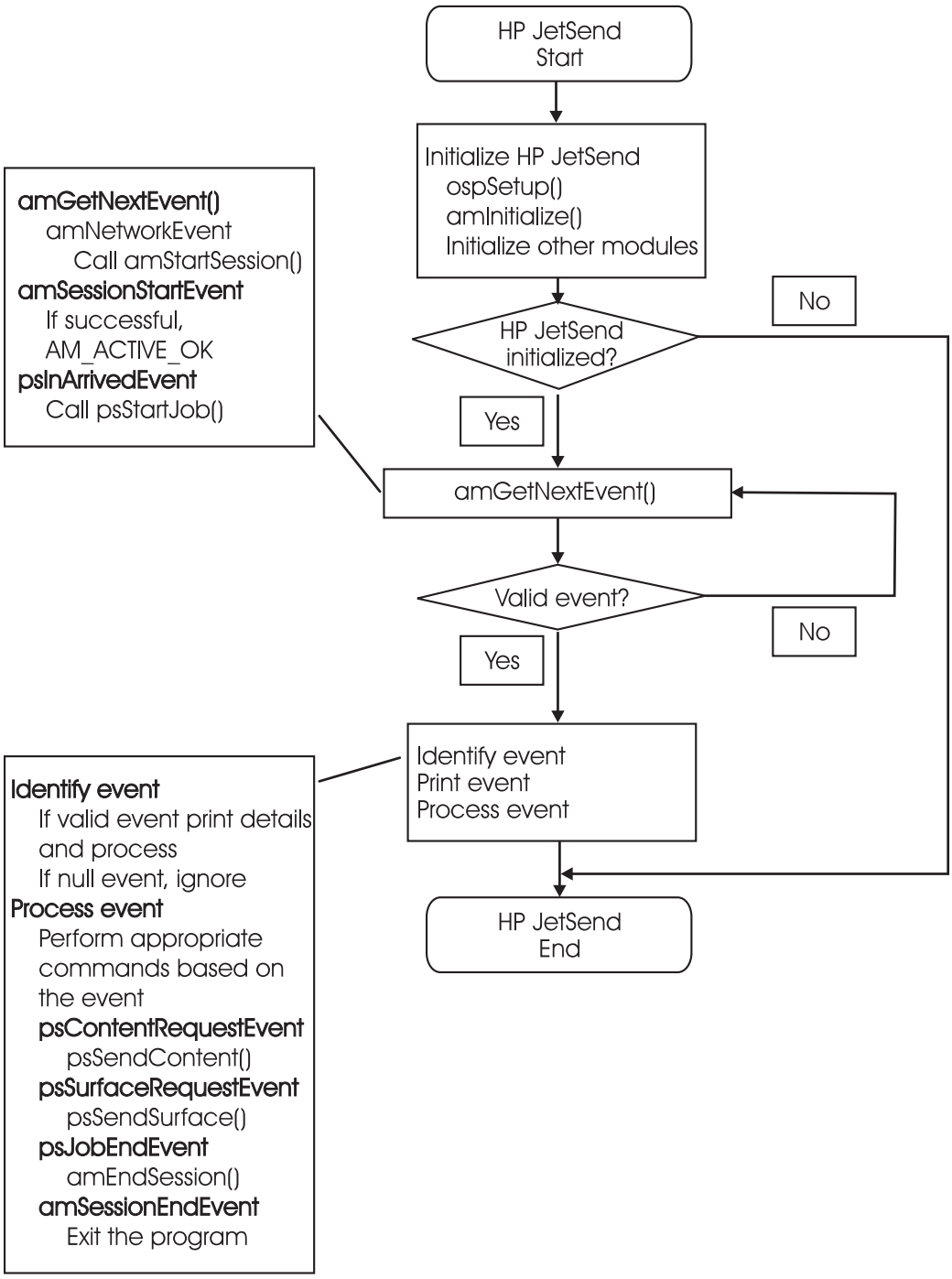
If the data sent in the last step was the description of a child, then wait for a `psSurfaceRequestEvent` message. It represents a request for the surface description of that child.

- If a `psSurfaceRequestEvent` message was received, then after the surface description has been sent, wait for a `psContentRequestEvent` message. It should represent a request for the content of the current child.

If a `psSurfaceRequestEvent` message and a subsequent `psContentRequestEvent` message are received, send the content data of the child.

- Repeat the above process. Start with waiting for the initial `psContentRequestEvent` message and continue until all the children's content data has been sent.
- After all the content data has been sent and the job has ended, the sender should receive a `psJobEndEvent` message.
- Call `amEndSession()` to end the session.
- Call `ospCloseDown()` to clean up the JSP layer properly.
- Exit the program. (This assumes a single run of the program.)

Figure 9.2 JetSend Sender Program Flow Chart



Common Sender Files

Commonfunc

This file contains functions for initiating sessions. It also provides functions to print various JetSend events.

Emparsefunc

This file contains general functions to parse e-material. Each of the major e-material encodings (image, text, file, plane, association) appears in a sample.

Getuserinput

This file contains routines to get inputs from the user. Input information includes text strings to send, the ip address of the remaining receiving device, and so forth.

Description of the Content Data Sending Function

The function for sending renderable content data is available for the fundamental encodings only. It can be found in all the sender modules except in InlineTextSender since in that program the content data is sent in-line with the surface description. The function is called `Send`*Encoding*`EmContent()`, where *Encoding* is replaced with Text, File, or Image. For example, in the image sender program, ImageSender, the function is called `SendImageEmContent()`. The function can be found in the file *encoding*emsender.c, where *encoding* could be text, file, or image. For example, the file *imageemsender.c* can be found in the module ImageSender. It can also be found in the AssocSender and PlaneSender modules because they send an image.

Virtually any amount of content data can be sent. The only limitations are:

- The sender should have enough memory for a temporary buffer to hold the data to be sent in each `psSendContent()` call. The maximum amount of data that can be sent in each `psSendContent()` call is represented by iMaxBufferSize, which is specified in the tops_device.h file.
- The amount of the receiver's available memory.

The function sends data starting from the location (***vStart*** value) specified by the receiver. Data is sent to the receiver by calling `psSendContent()`. The total amount of data sent to the receiver, per each `psContentRequestEvent` message, is less than or equal to the maximum amount of data requested. The maximum amount of data requested is represented by ***vNumber*** in the `psContentRequestEvent` message. However, it may take multiple `psSendContent()` calls to reach the maximum amount of data requested. This is because the maximum amount of data sent in each `psSendContent()` call can only be equal to or less than the value specified by `iMaxBufferSize` in file tops_device.h.

If the total amount of content data available is greater than the maximum amount of data requested, the receiver would have to make additional content requests for data, which would result in `psContentRequestEvent` messages at the sender, in order to get the remaining content data. So it can take multiple requests for content data, hence multiple `psContentRequestEvent` messages to send all the available data to the receiver. It is expected that the receiver will specify in each `psContentRequestEvent` message the correct starting location from which content data is to be sent.

Identify event

- If valid event print details and process

- If null event, ignore

Process event

- Perform appropriate commands based on the event
- psContentRequestEvent
- psSendContent()
- psSurfaceRequestEvent
- psSendSurface()
- psJobEndEvent
- amEndSession()
- amSessionEndEvent
- Exit the program

Notes on Receiver Sample Programs

This section introduces the sample programs used to demonstrate a simple receiver (such as a printer) using the JetSend Communications Technology protocol stack.

The receiver sample programs first initializes the JetSend protocol stack and also initializes some support modules. They then enter a loop where events received through the JetSend stack are first displayed to the screen. This shows the order in which the events are received and the detail of each event. Each event is then processed by a call to an event trigger function. Once the event is completely processed, the memory allocated to the event is returned to the system and the next event is retrieved from the stack. When the session with the sending device is ended, the JetSend protocol stack is shut down along with the support modules.

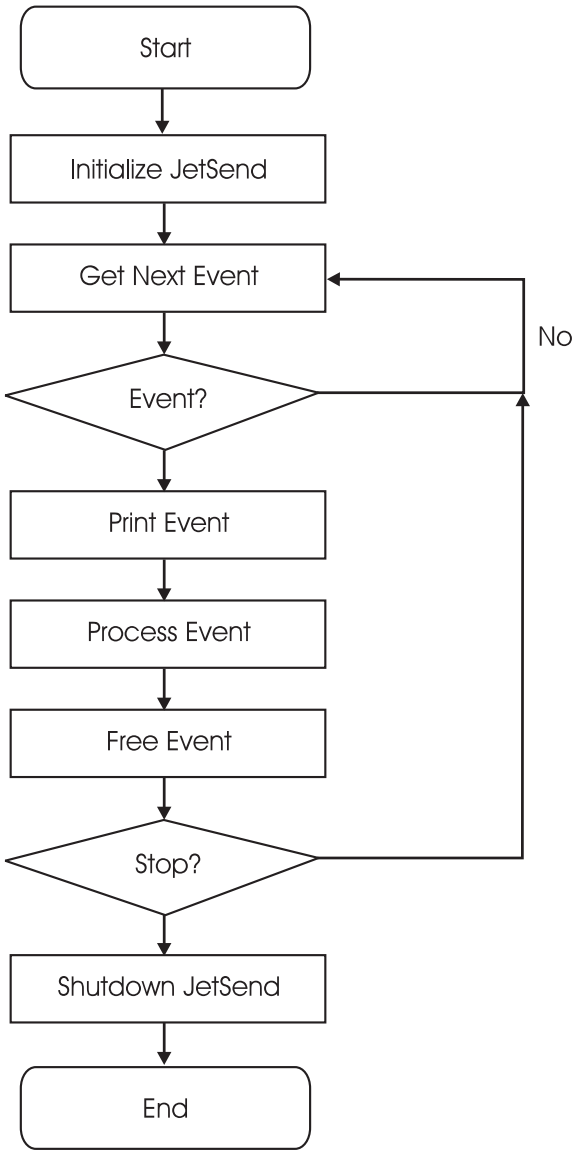
Note: There is very little error handling in the sample code. This was done in order to show the details of the algorithms used. Each device must implement its own error handling routines.

Note: These samples do not release JetSend surfaces. The surfaces are released when a job is ended or the session is closed.

Control Flow of the Sample Receiver Programs

Ths general algorighm used in the main processing loop for these examples is shown in the following figure:

Figure 9.3



This algorithm first initializes the JetSend protocol stack and also initializes some support modules. It then enters a loop where events received through the JetSend stack are first displayed on the the screen. This is done to show the order in which the events are received and the detail of each event. Each event is then processed by a call to an event trigger function. Once the event is completely processed, the memory allocated to the event is returned to the system, and the next event is retrieved from the stack. When the session with the sending device has ended, the JetSend protocol stack is shut down along with the support modules.

Common Receiver Files

All of the sample programs build upon the algorithms developed in the first sample “Hello, World!” program. They all share a set of common files that will handle the various bookkeeping needs of a JetSend receiver. These examples will start by working with a text encoding where the

data is not included in-line, but must be requested separately. Each new example will introduce a different encoding for the data being received.

Receiver

This file contains the main processing loop used in all of the sample receiver programs.

The function `initializeJetSend()` is used to set up the JetSend protocol stack and some support modules. It first calls `ospSetup()` to initialize some of the internal modules used in the JetSend protocol stack. This function must be called before any other calls into the JetSend stack. The Activity Manager is then started with a call to `amInitialize()`. This function takes the subaddress as its only parameter. Both the Contact Activity and the Push Receive Activity can now be initialized. The Contact Activity will allow the program to provide a text label of the receiving program to a sending device. It will also automatically get a label from the sending device. The Push Receive Activity will manage the process of receiving e-material from a sending device. Finally, the support modules are initialized. These modules are described below.

To shut down the JetSend protocol stack along with the support modules, the program calls `shutdownJetSend()`. This function simply shuts down each of the separate activities in the reverse order they were initialized.

When e-material has been sent to this program, the function `parseEMaterial()` is called in order to locate the renderable data. It begins by looking for the optional language attribute. If the data is offered in one or more languages, the **English language** (“en”) is selected. Since this attribute is at the null level, the language selected is applied to all the remaining selections. The function then looks for Text and ASCII attributes. (For this simple example, the Size attribute is ignored.) The text data could be at this level in the hierarchy, so a quick check is performed. If the data is not found, the program looks for the data level and extracts the text. This example simply displays the text data to the screen.

A few extra lines have been added to `initializeJetSend()` to provide preferred encodings and status to a JetSend sending device. By providing preferred encodings, a receiver can let a sending device know ahead of time what selections it will make if given the opportunity. The sending device can use this information to help create its e-material offerings. The function used to build the self-encodings is discussed below. The Status Activity is also initialized in order to provide both device and job status information to a sending device.

The preferred encodings for this receiver are created in the `buildSelfEncodings()` function. This function accepts an empty buffer and adds the encodings. This function must be updated as each new encoding is added to the examples.

StatusSupport

The first function in these files, `createLanguages()`, creates a structure that is passed to the Status Activity during its initialization. This structure is used to tell a sending device what languages and symbol sets will be used to report both device and job status. In this case, only the English language using the ASCII symbol set will be used.

The other two functions in these files, `setDeviceStatus()` and `setJobStatus()`, are simple wrappers around the JetSend functions `ssSetDeviceStatus()` and `ssSetJobStatus()`. These wrappers are used so that only one function call needs to be made to set both the status value and the status string for either the device or the job.

EventSupport

These files perform two important functions for the example receiver. The details of each event are displayed to the user through the `printEvent()` function. This function accepts an `AMEVENT` structure and prints the member detail to the screen depending upon the type of event.

The second operation these files provide is to maintain a linked list of pointers to event trigger functions. For each type of JetSend event, a "trigger" function is created. These trigger functions take the same number and type of arguments, in this case a pointer, to an `AMEVENT` structure. A pointer to each trigger function is then added to the linked list through `addNewEventFunction()`. When a JetSend event arrives, `processEvent()` is called with a pointer to the event. This function looks through the list of trigger functions until it finds a function that matches the type of event. If a trigger is found, that function is called with a pointer to the event. If no trigger function is found, a message is displayed to the user and the function exits.

This method of maintaining a list of trigger functions helps to work with the asynchronous nature of the JetSend events.

EventTriggers

These files contain the individual functions that will be called for each JetSend event that arrives for the receiving device. They also contain a structure (`SESSIONINFO`) that will be used to maintain information about the current JetSend session and sending device.

When a sending device starts a new session, the `amSessionStartTrigger()` function is called. Since this simple example only supports one session at a time, `amListenForSession()` is called in order to refuse any new sessions until this one is complete. The function then records the information about the current session and sending device for later use.

When either device ends the session, `amSessionEndTrigger()` is called to begin listening for new sessions again. It also clears the information about the last session and resets the device status to Ready. Finally, it clears the flag used in the main processing loop. This allows the program to shutdown the JetSend protocol stack and end the program.

If an error is detected on the network, `amErrorTrigger()` is called. For this simple example, no error recovery is performed. The function only tries to shutdown the JetSend protocol stack and end the program.

The function `conRemoteComponentTrigger()` is called when a component is received from the sending device. This component could include the sending device's self-label (for example, "John's scanner") or an icon representing the type of sending

device. If the component is a self-label, the information is stored in the SESSIONINFO structure.

The prInRejectedTrigger() function is only called if the sending device has rejected the well-known "In" surface of the receiver. This could mean that the sending device does not have a properly initialized push send module installed. In any case, it means that the local device will never receive any information from the sender so this function shuts down the open session by calling amEndSession().

When a sending device starts a new job, prJobStartTrigger() is called in response to a JetSend event. This trigger function sets both the device and job status to busy, adds the job-specific information to a linked list (see JobSupport below) and passes the job information to processJob() for e-material parsing.

If the sending device prematurely ends a job, the prJobEndTrigger() function is called. This function simply resets both the device and job status to "Ready" and removes all job information from the linked list.

The function prSurfaceTrigger() is called when a response to a request for surface detail has been received. This trigger is only used for association and plane encodings. If no information is available from the sending device, the function tries to end the job. Otherwise, it parses the reply to determine the type of reply and then calls the appropriate handling function.

After a response to data content has been received, prContentTrigger() is called. This function looks up the type of the original request and then passes a pointer to the e-material handling function.

If the sending device rejects the request for data content, prContentRejectedTrigger() is called. For this simple example, no determination is made for the rejection, and the job is ended.

When the Activity Manager is first initialized, it determines which networks are active. These results are sent back to the receiver via an amNetworkEvent. This event must arrive before the JetSend protocol stack can perform any work for the receiver. The amNetworkTrigger() function starts the process of listening for new sessions and records the type of network (for example, "ip:").

JobSupport

The JobSupport module maintains a linked list of JOBINFO structures. These structures are used to retain information about each current job. The information contained in each structure includes: the name of the top-level surface of the job, the job ID number, the handle number of the top-level document, the session handle number, and the number of outstanding requests that the receiver has placed to the sending device. When the number of outstanding requests has dropped to zero, it is safe to remove all information about that job and formally end the job with the sender.

When a sending device starts a new job, the information is stored into a JOBINFO structure by making a call to addNewJob(). This function allocates memory and fills the

members of the structure with the appropriate data. It then adds the structure to the list of jobs currently being processed.

The function `removeJobByNumber()` removes the job information structure from the linked list of jobs and frees any associated memory. This function is used to clear a job from the list of jobs.

When information about a particular job needs to be retrieved, the `findJobById()` function returns a pointer to a `JOBINFO` structure given a job ID number. It is important to note that this function does not remove the job from the list, but simply returns a pointer to the information contained in the node of the list.

The `cancelJob()` function is similar to `removeJobByNumber()` but it also removes all requests from their respective list, resets both the device and job status, and formally ends the job with the sender by calling `prEndJob()`. This function is used when a job needs to be ended and all information concerning the job also needs to be removed from all linked lists.

When the receiver completes each job related task, it tries to end the job through a call to `endJob()`. This function locates the job information structure in the linked list to determine if all outstanding requests have been completed. If there are no more outstanding requests, the job information is removed from the list, the job is ended, and the job status is reset. If there are no more jobs in process, the device status is also reset to Ready. This function performs no action if outstanding requests remain in the job information structure.

As each new buffer of e-material is received either because a new job is started or through a response to a content request, the buffer is sent to the `processJob()` function for parsing. This function first looks for the language encoding if it is offered. If the encoding includes a language specification, only “en” is supported in this example. The buffer is then parsed for one of the major encodings and the appropriate function is called. As each new encoding is added to the examples, this function will need to be updated.

RequestSupport

This module is similar in operation to the JobSupport module. It maintains a linked list of requests made to the sending device along with various functions used to retrieve information from the request structure. The structure `REQUESTINFO` is used to maintain the following information for each request: the request number assigned by the JetSend protocol stack, the current job number, an enumerated value for the type of request (such as text, file, or image), a flag to watch for the first block of content data, and a filename where the data content will be stored.

When a new request is sent to a JetSend sending device, `addNewRequest()` is used to add the `REQUESTINFO` structure to the linked list.

The following functions are used to retrieve information from a `REQUESTINFO` structure stored in the linked list; their use and return value are evident from the name of the function.

`findJobByRequest()`

`findReqTypeById()`

`findReqById()`

`removeRequestByNumber()`

`removeRequestsByJob()` (This removes all requests for a given job number.)

When the receiving device needs to request more data content from a JetSend device, a call is made to `prRequestContent()`. The function `requestContent()` is used as a wrapper around this JetSend function. One of its inputs is a buffer (currentSelections) that contains the selections made during the processing of the e-material sent by the sending device. An item is added to this buffer to instruct the sender to start at the beginning of the data, then this buffer is copied into a larger buffer and sent to the current sending device. The e-material that was sent is then displayed to the user for information purposes. A new request information structure is created to record the request made, and is then added to the request list. Finally, the number of outstanding requests for this job is incremented.

When the first block of data content is returned, it will contain a header. The function `printContentReplyHeader()` is used to display this header information to the user.

The function `requestSurface()` is similar in operation to `requestContent()`, but is used to request and record information specific to a JetSend surface.

Sample One Sender—InlineTextSender

InlineTextSender sends user-entered text to a receiver. It sends the text encoded as ***vData*** in-line with the encoding top-level surface description. Thus, in-line means the content data is sent with the surface description.

After all the initialization has been completed successfully and the psInArrivedEvent message has been received, the program requests the user to enter the text to be sent. The user-entered text is encoded as ***vData*** in the e-material description for the text encoding. The amount of data that can be sent in-line is limited to the amount of space left after the surface description has been written to the buffer. (See the code for details.) The buffer containing the e-material surface description and content data is sent to the receiver via a `psStartJob()` call.

If the transmission is successful, the sender receives a `psJobEndEvent` message, then calls `amEndSession()` to end the session. To shutdown the system properly, `amShutdown()` has to be called. After that, `ospCloseDown()` is called to reset the JSP layer and the program exits.

The vLanguage attribute value for “en” is a default value for ***vText***. This program offers only “en”.

The `vSymbolSet` attribute value ***vAscii*** is a default value for the ***vText*** encoding. `InlineTextSender` offers only the ***vAscii*** attribute and does not offer the ***vUnicode*** attribute although the JetSend system supports it. Thus, only ASCII text data is offered.

Sample One Receiver - Inline Text

Since every new programming idea starts with a simple “Hello, World!” program, this will be the first example. This program will receive only text that is sent in-line, i.e., the text data is received in the same packet as the initial `prJobStartEvent`.

RcvText1

This file contains the main processing loop, and is fully described above.

RcvTriggers1

These files contain the trigger functions that handle each type of JetSend event (see above). The first trigger normally called is `amNetworkTrigger()`. Calling this trigger means that the Activity Manager has detected an active network of some type. This function calls `amListenForSession()` if the network type is “ip:” to allow a sending device to initiate a session.

When a remote device attempts a connection, the `amSessionStartTrigger()` is called. Solely for bookkeeping reasons, this program works with only one remote device at a time. To stop other devices from attempting to connect, `amListenForSession()` is called with the second parameter set to FALSE.

When the current session is ended by either device, `amSessionEndTrigger()` is called. This function once again calls `amListenForSession()` to allow another device to initiate a connection. It also clears a global flag used to exit the main processing loop and shut down the program.

After a remote device starts a session, it will attempt to start a job. This signals that the sender has some data to be rendered on the receiving device. At this point, `prJobStartTrigger()` is called. This function passes a pointer to the e-material contained within the event structure to the `parseEMaterial()` function discussed earlier. After the e-material has been interpreted, a call to `prEndJob()` is made to tell the sending device that the job is complete. It is now up to the sending device to end the session.

Sample Two Sender—TextSender

TextSender is similar to InlineTextSender, described above. After the `psInArrivedEvent` message has been received, the TextSender sends the job's encoding surface description in a `psStartJob()` to the receiver. However, instead of sending the content data (user-entered text) in-line with the surface description, it sends the content data after it receives a `psContentRequestEvent` message.

The ***vLanguage*** attribute and its value, "en", is a default attribute-value encoding for ***vText***. TextSender offers only "en".

The ***vSymbolSet*** attribute value ***vAscii*** is a default value for the ***vText*** encoding. TextSender offers only the ***vAscii*** attribute and does not offer the ***vUnicode*** attribute although the JetSend system supports it. Thus, only ASCII text data is offered.

Unlike InlineTextSender, which is limited by the available space in the description buffer, TextSender can send virtually any amount of data to the receiver. A listing of the function that sends the content data can be found in file textemsender.c. However, a description of how the function that sends content data works can be found at the end of this section.

Sample Two Receiver—An Improved Text Receiver

It is now build a better receiver for text type e-material, using the common files described above. This new receiver will be able to accept text in-line or through a separate content request. It is able to accept multiple jobs and provide status feedback.

Changes to the Common Files

In order to provide as much feedback to a JetSend device as possible, the preferred encodings for this receiver will be built in the function `buildSelfEncodings()`. These encodings will be Language = English, Encoding = Text, Size (don't care), and Symbol Set = ASCII. These encodings will be provided through the Contact Activity.

A 'case' statement needs to be added to the `prContentTrigger()` function. Recall that this function will be called when a response is received from a previous content request. When the reply is a result of a request for text content, the function to be called to parse the e-material will be `processTextContent()`.

When a new job is started, the e-material contained in the `prJobStartEvent` will be sent to the function `processJob()`. An "if" block needs to be added to this function after the language encoding has been detected. This block will test for the presence of text e-material. If this type of encoding is offered then the e-material buffer will be passed to `processTextEncoding()`.

The last addition to the set of common modules is to add a member to the enumerated `ReqType`. This will be used to record the type of content request that may need to be made.

Encoding Module

The text module contains the two specific functions used for text e-material. The function `processTextEncoding()` is called to examine the initial e-material buffer included with the job start event. It simply walks through the e-material looking for ASCII encoded text data. It records all selections made during the search and uses these selections to request the text data if it was not found in-line. If a content request is made the corresponding reply is sent to `processTextContent()`. This function displays the header information contained in the first block and displays the text to the screen.

Sample Three Sender—FileSender

As with the TextSender, the FileSender does not include the file data in-line with the encoding surface description sent to the receiver. The file data is sent after a psContentRequestEvent message has been received. Nevertheless, the JetSend system allows the file content data to be sent in-line if there is sufficient space in the description buffer after the description has been written to the buffer.

In FileSender, although ***vLanguage*** is optional, “en” is offered. Two other optional attributes, ***vMimeType*** and ***vChildIcon*** , are not offered. Please see the E-Material Specifications in this ADK for a description of these attributes. ***vChildIcon*** provides an icon representation of the file to the receiver. The icon could be an image but on a separate (child) surface and so should be encoded accordingly, as ***vImage*** (if it is an image). The ImageSender program shows how to send an image.

Since all the attributes under the ***vFile*** encoding are optional, there are no default or base attribute values for the ***vFile*** encoding.

Only the actual file binary data, that is, the file without the header, is sent to the receiver. A listing of the function that sends the content data can be found in file fileemsender.c. However, a description of how the function sends the content data can be found earlier.

Sample Three Receiver—Receiving Files

Files are one of the easiest encodings to handle, since all of their attributes are optional. This capability will be added to the current text receiver.

Changes to the Common Files

The `buildSelfEncodings()` function in the Receiver module needs to have another encoding added to it. The file encoding will be placed before text encoding in the list of encodings. This order tells the sending device which encoding the user prefers to receive, if given a choice. Since files encodings are simpler than text, file will be placed first in the list.

The trigger function `prContentTrigger()` will have a case statement block added to it to direct the e-material buffer from a file content request. In this case, the buffer will be sent to the handling function `processFileContent()`.

Likewise, the 'if' block that is added to the `processJob()` function in JobSupport will be placed before the text block. This is where the file selection is made over text if given the opportunity. If the job data is offered as a file then the e-material buffer is passed to `processFileEncoding()` for parsing.

Finally, another member is added to the enumerated `ReqType` in the `RequestSupport` module. This will be used to flag requests for file content.

Encoding Module

The File module contains the two functions used to handle file type e-material. The function `processFileEncoding()` is called to handle the initial e-material buffer. This function is similar in operation to that used to handle text e-material. Although this function ignores the values for the MimeType, DataSize, and ChildIcon attributes, it adds the values to the current selection list if they are present. If a filename is present in the e-material, it is used to store the file data; otherwise the default filename of "JetSendFile.dat" is used. This function stores the file data if it is sent in-line (for a small file) or makes a content request with the current selection list. A reply to the request is handled by `processFileContent()`. This function skips the header information in the first reply block and stores the data in the appropriate file. A comment is displayed to the user when the last block of data is received.

Sample Four Sender—ImageSender

The image encoding is much more involved than the text and file encodings. In fact, the image encoding is the most complex of the fundamental encodings. There are a lot more offerings and selections for the receiver to choose from than in text or file encoding.

Any device that supports images is required to offer or accept the image default encoding. In addition, it is recommended that such devices also offer or accept the image base encodings. The default encoding for images is 1 bit gray (monochrome) at 300 dpi and RLE compressed. The base encodings for images is 8 bit gray at 150 dpi with no compression and 8 bit gray at 150 dpi with RLE compression. An additional base encoding for color images is SRGB 24 bit color 150 dpi with no compression.

A requirement to offer the default encoding means that regardless of the other formats of the image to send to the receiver, a 1 bit monochrome 300 dpi RLE compressed version of the image must also be offered. Thus, the receiver can choose the original format of the image and the default encoding format. This ensures that JetSend devices supporting images can send an image that the receiver is capable of rendering. If the base encodings are supported, they allow the devices to transmit and receive the image at a higher fidelity than that of the default encoding.

For example, if a sender wants to send an 8 bit gray 300 dpi uncompressed image, it will also have to offer the choices of 8 bit gray 150 dpi RLE-compressed and 1 bit mono 300 dpi RLE-compressed. This assumes that the sender is offering the base encodings in addition to the default encodings. Thus, based on the capability of the receiver, it could select the original image being offered, or either one of the two additional default and base encodings.

In another case, if a user wants to send a 24 bit RGB (color) 150 dpi JPEG-compressed anticedent image, it would also have to offer thedefault and base encodings of the color image. These additional offerings would be a 24 bit SRGB (color) 150 dpi uncompressed image, a 8 bit gray 150 dpi uncompressed image, a 8 bit gray 150 dpi RLE-compressed image, and a 1 bit mono 300 dpi RLE-compressed image.

The ImageSender offers all the default and base encodings. The E-Material Specifications details offerings available for all the encodings.

After a `psContentRequestEvent` message is received, the image's content data is sent based on the receiver's selections. Image test files are provided for the ImageSender. These test files are described below.

Test files

The ImageSender works only with the two sets of image test files provided with the program. However, the program could be modified to work with other image files. One file contains an 8 bit gray 150 dpi uncompressed image and the other file contains a 24 bit SRGB (color) 150 dpi uncompressed image. (*Actually the second file contains an RGB image.)* Default and base encoding formats of the original files have also been provided. The original files are Windows BMP format, and though the default and base encoding format files maintain the BMP header, the RLE compression is as described in the E-Material Specification. Also, because the BMP format

has the origin of the image in the lower left corner, the data has been inverted in the files so that the origin is at the top left corner.

Program Flow

After the JetSend system has been initialized and the `psInArrivedEvent` message has been received, the user is asked to select one of the two test files. The first file contains an 8 bit gray 150 dpi uncompressed image. This file represents a base encoding for image. The second file contains a 24 bit SRGB (color) 150 dpi uncompressed image. This file represents one of the base encodings for color devices.

The file header and image header information are extracted from the file selected by the user, and the information is used to build the surface description for the encoding.

If the user chooses the first file, 8 bit gray 150 dpi with no compression, the function `BuildGray8Bit150NoneDesc()` builds the top-level surface description. In addition to offering the selected file, the base and default encoding formats of the image are also offered. The base format is 8 bit gray 150 dpi with RLE compression and the default format is 1 bit gray (mono) 300 dpi resolution with RLE compression. The details of how this top-level surface description is built can be seen in the function in file imagedescription.c.

If the user chooses the second file, 24 bit SRGB (color) 150 dpi resolution with no compression, the function `BuildSRGB24Bit150None()` builds the top-level surface description. In addition to offering the selected file's color image, the base gray and default mono formats are also offered. As stated previously, these are 8 bit gray 150 dpi resolution with RLE compression, 8 bit gray 150 dpi resolution with no compression, and 1 bit gray (mono) 300 dpi resolution with RLE compression.

After the receiver's selections are returned to the sender in a `psContentRequestEvent` message, the Sender sends the requested content data based on the receiver's selections. To send the requested image's content data, the Sender extracts the file and image header information from the file. With this information, the Sender reads a chunk of the data into a buffer and sends it. This process continues until all the data has been sent or until the maximum amount of data requested by the receiver is reached. If the maximum amount of data requested by the receiver is reached before all the data is sent, the receiver will have to make additional content requests to get the remaining data. Virtually any amount of data can be sent to the receiver.

The process of the function that sends the content data has been explained previously.

Sample Four Receiver—Receiving Images

Images have the most complex encodings of the currently implemented "simple" encodings. This is due to the number of options and different formats that can be used to render an image.

Changes to the Common Files

The image encoding will be added to `buildSelfEncodings()` in the Receiver module. This encoding is placed after the encodings of file and text (the reasons for this are explained above). The attributes and values to be added for this encoding will be the default encodings of Size (don't care), ColorSpace = Gray, PixelDepth = 1, Resolution = (300, 300), Pixels = (2400, 3000), and Compression = RLE.

A case statement block is added to `prContentTrigger()` in the EventTriggers module to handle replies to requests for image data. These replies will be sent to the function `processImageContent ()`.

Another "if" block is added to the `processJob()` function in the JobSupport module. This block will direct the initial e-material buffer in the `prStartJobEvent` to the `processImageEncoding()` function for parsing.

Two changes need to be made to the RequestSupport module. First, a member needs to be added to the enumerated ReqType to flag requests for image content data. Second, the attribute added to a content request buffer in `requestContent()` to specify the starting position of the data needs to be modified for an image request. Instead of using the attribute ***vStart*** to specify the starting position, images use the ***vStartByte*** attribute. In either case, the attribute is set to zero to indicate that the receiver wishes to start at the beginning of the data.

Encoding Module

The Image module contains functions similar to the previous two modules for file and text. However, due to the complexity of the image encoding, the `processImageEncoding()` is slightly longer. The function ignores the values for the ***vSize*** and ***vOpacity*** attributes, but adds these values to the list of current selections. In this simple example, only the default encoding attributes are selected. These attributes are listed above under the description to `buildSelfEncodings()`. To keep this example program simple, only the raw image data is stored in the file JetSendImage.raw. If the image data is not included in-line, a request is made with the resulting reply being sent to `processImageContent()`. This function skips the header information in the first data block and stores the raw image data in the above file.

Sample Five Sender—PlaneSender

The PlaneSender has two child surfaces, one with a fundamental encoding of ***vText*** and the other of ***vImage***. The sender first sends a top-level surface description specifying that there are children, but does not specify how many of them there are. The sender also specifies that the children are to be processed sequentially. This means that processing starts with the first child and is completed for that child when the content data for that child has been sent to the receiver. Processing of the next child then starts and is completed before the next child is processed. Also, any data can be requested for just one child at a time. This continues until all the children have been processed.

Since a plane may have a front side and a back side, the receiver can choose to process either of them first. Sequential processing means that if, for instance, the receiver requests data for the back side first, all children on the back side must be processed before any processing can be initiated or completed for the front side. Conversely, if the receiver requests data for the front side first, all processing of the front side's children must be completed before any processing of the back side is initiated.

The following assumes that the front side is processed first.

After the top-level surface description is sent to the receiver, the sender receives a first `psContentRequestEvent` message representing a request for the first child's content information (top-level description) with the ***vFrontFirst*** attribute. This content information could have been included with the initial top-level surface description by specifying the number of children available and their names. Further, if the number of children and their names had been specified, the receiver could request the content information of more than one child at a time. However, surface description and content data can be requested for only one child at a time.

After receiving the content request for the child content information, the sender sends the name of the first child. Using this name, the receiver requests the surface description of this child. For this request the sender receives a `psSurfaceRequestEvent` message. In response, the sender sends the first child's surface description which, in this example code, is for ***vText*** encoding. In this example code, the text is included in-line.

Although the content data is sent in-line, the receiver may still make its selections from the offerings and make a request for the first child's content data, resulting in the sender receiving a `psContentRequestEvent` message. The sender then sends the content data of the first child.

Using the name of the first child as a reference, the receiver requests the content information of the next child and the processing described above is repeated. The second child in this example is an image, and its data is not sent in-line with its surface description.

After processing of the front side children is completed, the processing of the back side children can begin.

The sending of the content data appears; it applies to the sending of text, file, and image content data.

Sample Five Receiver—Receiving Planes

Planes will be the first "complex" encoding to be added to the receiver program. This complexity arises from having child surfaces. This surface hierarchy allows complex documents to be represented.

A plane surface can have both a front and back side with multiple children contained on each side. The children will be requested sequentially starting with the front side of the plane. It is also possible to have no children to a plane. In this case, the attributes of the plane contain the information being transferred.

Changes to the Common Files

The Plane encoding will be added to the list of encodings this program would prefer to receive in the `buildSelfEncodings()` function. The ***vChildFront*** and ***vChildBack*** attributes are set to –1, Sequential. This means that the receiver would prefer to handle the children one at a time in a sequential order. The ***vBackCoord*** and ***vOpacity*** are both set to zero, and the attributes ***vSize*** and ***vColor*** are also set to zero.

The case block is added to the `prContentTrigger()` function in the `EventTriggers` module. This case block will call `processPlaneContent()` when the program receives a reply to a data content request from a plane encoding. The `prSurfaceTrigger()` function is called whenever the program receives a reply to a request for another surface. A case statement is added to this function to pass the reply data buffer to `processPlaneSurface()` for further parsing.

Just like the previous encoding types, an "else if" block is added to the `processJob()` function in the JobSupport module. This block directs the initial incoming e-material buffer to `processPlaneEncoding()`. Also, a check is performed at the beginning of the `endJob()` function to ensure that the next child in the plane is processed. This check is performed by calling `processNextPlaneChild()`.

The only change to the RequestSupport module is the addition of two members to the enumerated `ReqType`. One member will be used to flag requests for content information on planes and the other will flag requests for the plane's child surfaces.

Encoding Module

Two flags are maintained to determine if the receiver is currently working on the front or back of the plane. Also, if there are no children in the encoding, the other side of the plane is examined for information.

If a child is located on the front or back of a plane, a request for content is sent to the sending device. The reply to that request is handled by `processPlaneContent()`. The main purpose of this function is to retrieve the name of the child and request the surface of that child. It also stores information necessary to request the surface of the next child at a later time.

The function `processPlaneSurface()` works similarly to the function `processAssociationSurface()` described above. Also,

`processNextPlaneChild()` performs the same actions as `processNextAssocChild()` did for associations.

Sample Six Sender—AssocSender

The AssocSender module has two children of fundamental encodings, vFile and vImage. The processing of the association encoding is quite similar to the processing of the plane encoding. There are minor differences due mainly to the fact that the association encoding does not have a front side and a back side; it just has children, not front side children and back side children. Thus the first child's attribute is ***vFirst***, not ***vFrontFirst*** or ***vBackFirst***. With that in mind, the description of association encoding processing follows.

The AssocSender first sends a top-level description specifying that there are children, but does not specify how many children there are. The sender also specifies that the children are to be processed sequentially (***vSequential***). This means that processing starts with the first child (***vFirst***) and is completed for that child when the content data for that child has been sent to the receiver. Processing of the next child begins and is completed before the following child is processed. Also, any data can be requested for only one child at a time. This continues until all the children have been processed.

After the top-level surface description is sent to the receiver, the sender receives a first `psContentRequestEvent` message representing a request for the first child's content information (top-level description) with the vFirst attribute. This content information could have been included with the initial top-level surface description by specifying the number of children available and their names. Further, if the number of children and their names had been specified, the receiver could request the content information of more than one child at a time. However, surface description and content data can be requested for only one child at a time.

After receiving the content request for the first child's content information, the sender sends the name of the first child. Using this name, the receiver requests the surface description of the child. For this request the sender receives a `psSurfaceRequestEvent` message. In response, the sender sends the first child's surface description which, in this example code, is for vFile encoding.

After the receiver has made its selections from the offerings it makes a request for the first child's content data, resulting in the sender receiving a `psContentRequestEvent` message. The sender then sends the content data of the first child.

Using the name of the first child as a reference, the receiver requests the content information of the next child and the processing described above is repeated. (However, the second child in this example is an image.)

The sending of the content data is described above. It applies to the sending of text, file, and image content data.

Sample Six Receiver—Receiving Associations

The receiver will only process one association encoding at a time, to keep the sample simple. It will also always request content for the first child of an association encoding. This content will contain the name of the surface for the child. When the child surface is received, the program will ask for the data content of the first child and store the information necessary to request the second child surface. This information is stored because the receiver assumes that the children of the association must be processed sequentially. Although this algorithm places a slightly greater load on the network, it simplifies the logic of decoding the e-material.

Changes to the Common Files

The Association encoding will be added to the list of encodings this program would prefer to receive in the `buildSelfEncodings()` function. The ***vChild*** attribute is set to "***–1 vSequential***". This means that the receiver would prefer to handle the children one at a time and in a sequential order.

The case block is added to the `prContentTrigger()` function in the EventTriggers module. This case block will call `processAssiciationContent()` when the program receives a reply to a data content request for an association. The `prSurfaceTrigger()` function is called whenever the program receives a reply to a request for another surface. This function uses a select–case method to pass the reply data buffer to `processAssiciationSurface()` for further parsing.

Just like the previous encoding types, an "else if" block is added to the `processJob()` function in the `JobSupport` module. This block directs the initial incoming e-material buffer to `processAssociationEncoding()`. Also, a check is performed at the beginning of the *endJob()* function to ensure that the next child in the association is processed. This check is performed by calling `processNextAssocChild()`.

The only change to the `RequestSupport` module is the addition of two members to the enumerated `ReqType`. One member will be used to flag requests for content information on associations and the other will flag requests for the association's child surfaces.

Encoding Module

The `processAssociationEncoding()` function in the Association module works similarly to the encoding processing functions above, with the following differences.

- A check is performed to insure that only one association encoding is being handled at a time.
- If there are no children to the association, the function tries to end the job.

While processing the reply to a content request in `processAssociationContent()`, a check is performed to see if the number of children in the reply is zero. This would mean an end to the number of children in the association, and the flag used in `processNextAssocChild()` can be reset. The surface of the child named in the content reply is requested and information is stored in preparation for requesting the next child.

When a child's surface is returned from the sending device, the function `processAssociationSurface()` simply removes the request information from the stack and sends the e-material buffer to `processJob()`. Here it is handled as a regular surface where the e-material is parsed and sent to the appropriate handling function.

Before a job is ended, `endJob()` calls `processNextAssocChild()` to determine if the receiver is still processing an association. This call also initiates a request for the next child in the association.

Chapter 10

Testing Your JetSend Appliance

Introduction

Testing is a critical step in the development cycle of your JetSend appliance. This testing process will be more or less complicated, depending on the type JetSend appliance being designed. For instance, if JetSend technology is to be only one of the many features of the appliance, then testing will deal with both JetSend technology and the interaction it has with the remaining feature-set. The purpose of this discussion is not to provide a detailed test-plan for use in the testing-phase of a particular appliance implementation. Rather, a series of suggestions describing test-design will be presented for consideration.

A Three-Stage Approach

One way to approach JetSend appliance testing is to break it down into three stages. The first stage tests the JetSend core code in its simplest state. The second stage tests device-specific code and its interaction with JetSend core. The third stage tests a particular JetSend appliance as a single entity against a second JetSend appliance. This third and last stage of testing will only be possible if there is access to an existing appliance that uses the JetSend technology in a complementary way. For example, a person designing a JetSend scanning device would need a JetSend-enabled rendering appliance to test against.

Testing JetSend Core in the Target Environment

The primary purpose for testing JetSend core code is to verify its successful port/build in your development environment. The JetSend core code provided to you has been thoroughly tested in the HP-UX environment. Both white-box and black-box testing methods have been utilized in testing for API functionality, error condition behavior, and for proper memory use/management. Customers of this ADK are not expected to fully re-test JetSend core code. However, since there may be porting issues specific to JetSend core and a specific development environment, proper JetSend core functions on the system must be verified.

In order to test the functionality of JetSend core, a working source-code for both TIAPI and RMTP is needed. The reason for this is that JetSend core needs a working network layer in order to function. A reference implementation of both TIAPI and RMTP have been included with this ADK.

Test 1

The simplest test of JetSend core is to simply initialize JetSend core and then exit. If initialization is successful, the result will be at least one `amNetworkEvent` being generated by JetSend core. If system has been built using the provided TIAPI, only one `amNetworkEvent` will be received, since the reference implementation only supports the ip-protocol.

The following code sample illustrates one way to perform this test.

```
#include <stdio.h>
#include <stdlib.h>

#include "appl.h"
#include "tops.h"
#include "tops_device.h"
#include "tops_osp.h"

main()
{
     int continue_processing = 1;
     AMEVENT  *incoming_event;

     /* We first need to allow the JSP layer to initialize its
        internal memory/data as well as initialize the TIAPI
        layer. */

     ospSetup();

     /* The next call will initialize TOPS and give JSP our
        "subaddress" of 0. */

     if(amInitialize(0))
     {
        /* This means the call FAILED. We'll exit. */
           printf("Call to amInitialize() FAILED\n");
           exit(-1);
     }

     while(continue_processing)
     {
        /* We need to call amGetNextEvent() in order for the control-
           thread to be passed on through each layer of TOPS and out to
           the network-layer. The function amGetNextEvent() calls
           oipRun() which calls ospRun() which calls xptRun(). In this
           way each layer of TOPS gets a chance to process. */
```

```
    amGetNextEvent(&incoming_event);
    if(incoming_event)
    {
        /* If amGetNextEvent() returns an event, we need to
           switch on the event-type since we need to respond
           differently for each type of event. */

        switch(incoming_event->iEvent)
        {
            case amNetworkEvent:
                printf("\n-------  RECEIVED EVENT -----\n");
                printf("    event:  %d\n",
                    incoming_event> iEvent);
                printf("pNetworks:  %s\n",
                    incoming_event->data.amNetwork.pNetworks);
                printf(" pAddress:  %s\n",
                    incoming_event->data.amNetwork.pAddress);
                continue_processing = 0;
                break;
            default:
                printf("\n-------  RECEIVED EVENT -----\n");
                printf("    event:  %d\n",
                    incoming_event->iEvent);
                  break;
        }

        /* After "processing" the event, we need to free the data
           associated with it. To do this, simply pass it into the
           amFreeEvent()function. */

        amFreeEvent(incoming_event);
    }
}

/* Close-down TOPS */

if(amShutdown())
{
    /* This means the call FAILED. WeÌll exit. */
        printf("Call to amShutdown() FAILED\n");
        exit(-1);
}
```

```
}       /* END of function main() */
```

Successful execution of the above program should generate an amNetworkEvent containing valid data that will be displayed in the following manner:

```
>>prog

-------  RECEIVED EVENT -----
    event:  30
pNetworks:  ip:
pAddress:  ip:your.local.ip.address#0
>>
```

Execution of the above program will cause activity throughout each layer of JetSend core; in through the Device API, down through JetSend core, out through the TIAPI, back in through TIAPI, back up through JetSend core, and finally out through the Device API.

If the execution is not successful, you may have to interrupt the program manually since our code allowed for no user interaction. Possible reasons for failure include a problem with the network layer or a problem with the port of JetSend core to your environment.

If there is a problem with your network layer, most likely you will not receive the `amNetworkEvent`. It is the network layer that provides to JetSend core the information contained within the event. If JetSend core does not receive this information, it will not generate the `amNetworkEvent` and therefore the above program will loop continuously around the call to `amGetNextEvent()`.

A problem with the port of JetSend core to your environment might manifest itself in the above test in three ways. The first would be a “premature” exit of the program caused by a failure of one of the API calls. The second would be the lack of an `amNetworkEvent` being generated, which in this case will result in a continuous loop around `amGetNextEvent()`. The third would be incorrect data within the event; either garbage or NULL content.

JetSend Core Logging

In the event of a failure, you may want to turn on the logging capability of JetSend core in order to track down the problem. To do this, simply add the following four lines of code to the above sample:

```
.
.
#include “tops_device.h”
#include “tops_osp.h”

extern Tu32 giDebugLevel;
extern Tu32 giDebugModule;

main()
```

```
{
int continue_processing = 1;
     Tstatus   status;
     AMEVENT  *incoming_event;

     giDebugLevel = ENTRY|INFO|WARNING|FATAL|FREQUENT;
     giDebugModule = LOG_ALL;

.
.
```

After recompiling with the additional four lines, you can execute the program to generate a log file. The two global variables are declared in the tops_log.c module and are used by JetSend core to determine the amount of logging as well as which modules will create log statements. The above assignment to `giDebugLevel` enables JetSend core to create log statements for each of the five possible logging levels. By assigning LOG_ALL to `giDebugModule` you enable every module of JetSend core to create log statements. It is possible for you to choose any combination of logging levels and modules if you simply OR your selections together. Valid selections for the above variables are given below.

giDebugLevel

ENTRY	All function entry-points as well as some exit-points.
INFO	Helpful information generated within functions
WARNING	Possible internal errors, not necessarily fatal.
FATAL	Critical internal errors.
FREQUENT	Frequently generated log statements.

giDebugModule

LOG_ALL	Allow every module to create log statements.
LOG_RMTP	Allow the RMTP modules to log.
LOG_LINK	Allow the TIAPI modules to log.
LOG_OSP	Allow the JSP modules to log.
LOG_OIP	Allow the JIP modules to log.
LOG_AM	Allow the AM module to log.
LOG_ACT	Allow all “Activity” modules (con, pr, ps, etc.) to log.
LOG_LIBS	Allow Library modules (like appl_emlib.c) to log.

By default, the makefile has -DLOG_TO_FILE defined, which causes all of the log statements to be written to a file. The file will be created in the execution directory and will default to the name `log__`*Day:hour:min:sec*. Be aware that selecting FREQUENT for `giDebugLevel` may result in extremely large log files, especially if your application is simply looping around a call to `amGetNextEvent()`. If you remove the DLOG_TO_FILE from the makefile, every log statement generated by core will be displayed on the terminal.

Listed below is a portion of the log-file generated by the above code-sample:

```
LINK> Entering xptInitialize.
RMTP> tiapiSocketsInitialize
RMTP> RplyModuleInitialize
RMTP> rmtpInitialize
RMTP> rmtpAddrInfoModuleInitialize
RMTP> rmtpDmnInfoModuleInitialize
RMTP> rmtpConnModuleInitialize
RMTP> rmtpConnIDInitialize
LINK> Entering xptEnumerate.
OSP > -ospChangeDefaultSubAddr newSubAddr[0]
OSP > -end ospChangeDefaultSubAddr retval[0x0]
AM  > amInitialize()
AM  : trigger: 36
AM  : session: 56
OIP > oipInitialize(0,40011df0)
OSP > -ospInitialize oipHandle[0]
OSP > -end ospInitialize status[0x0]
OIP > oipGetNextHandle()
OIP > oipLinkSurface(40017780)
OSP > -ospEnumerateNetworks: oipHandle[0] gateway_request[0]
OSP > -end ospEnumerateNetworks status[0x0]
OIP # oipRun(7b033ffc)
LINK# Entering xptRun.
\
```

The first part of each log statement identifies the module in which the function generating the statement can be found. The second piece of information signifies the level of the given statement. The rest of the data given is simply that which the function has been programmed to provide. For example, the statement

```
OIP > oipInitialize(0,40011df0)
```

can be broken down into its various components. "`OIP`" signifies that the log statement was generated in the oip module. The ">" identifies the statement as having a logging level of type ENTRY. The remaining information states that the function oipInitialize() has been called with input parameters of 0 and memory location 40011df0.

In addition to the ">"character, there are others used by the logging mechanism to relate information. For instance, the "#" represents the FREQUENT level and the ":" represents the INFO level. The two remaining symbols not shown above are "?" which means WARNING and "!" which means FATAL.

For a description of the logging mechanism, see both appl_log.h and appl_log.c. These files describe in detail the internal logging mechanism as well as the output format.

Test 2

By adding a small amount of code to the above sample, a second test can be written that will test JetSend core to a significantly greater degree. This test will actually cause JetSend core to make a connection with itself resulting in more segments of JetSend core code being hit. The example given below contains everything from Test 1 as well as some additional code (written in bold typeface).

Test 2:

```
#include <stdio.h>
#include <stdlib.h>

#include "appl.h"
#include "tops.h"
#include "tops_device.h"
#include "tops_osp.h"

main()
{
     int       continue_processing = 1;
     int       session_start_count = 0;
     int       session_end_count = 0;
     AMEVENT  *incoming_event;

     /* We first need to allow the JSP layer to initialize its
        internal memory/data as well as initialize the TIAPI
        layer. */

     ospSetup();

     /* The next call will initialize TOPS and give JSP our
        "subaddress" of 0. */

     if(amInitialize(0))
     {
     /* This means the call FAILED. WeÌll exit. */
```

```
        printf("Call to amInitialize() FAILED\n");
        exit(-1);
}

    while(continue_processing)
    {
    /* We need to call amGetNextEvent() in order for the
       control thread to be passed on through each layer of
       TOPS and out to the network-layer. The function
       amGetNextEvent() calls oipRun() which calls ospRun()
       which calls xptRun(). In this way each layer of TOPS
       gets a chance to process. */

       amGetNextEvent(&incoming_event);
       if(incoming_event)
       {
       /* If amGetNextEvent() returns an event, we need to
          switch on the event-type since we need to respond
          differently for each type of event. */

          switch(incoming_event->iEvent)
          {
             case amNetworkEvent:
                printf("\n-------  RECEIVED EVENT -----\n");
                printf("    event:  %d\n"incoming_event>iEvent);
                printf("pNetworks:  %s\n",
                incoming_event->data.amNetwork.pNetworks);
                printf(" pAddress:  %s\n",
                incoming_event->data.amNetwork.pAddress);

             /* Now that we've received the amNetworkEvent for ip,
                we can make the call to amListenForSession() and
                wait for connection over the ip: network. */

                if(amListenForSession("ip:", 1))
                {
                /* The call returned an ERROR.  Abort. */
                   continue_processing = 0;
                }
                else
                {
                /* If the call succeeds, we can next issue a call
                   to amStartSession() and pass in our own ip address. In
```

```
                this test-case, we'll simply use the pAddress member
                of the amNetworkEvent
                we've just received because we know it contains only our
                ip address. This should result in TOPS connecting to
                itself and generating two amSessionStartEvents. */

                    if( amStartSession(incoming_event-> \
                        data.amNetwork.pAddress))
                    {
                      /* The call returned an ERROR. Abort. */
                        continue_processing = 0;
                    }
                }
    break;
              case amSessionStartEvent:
    printf("\n-------  RECEIVED EVENT -----\n");
    printf("    event:  %d\n", incoming_event->iEvent);
    printf("  iStatus:  %d\n",
        incoming_event->data.amSessionStart.iStatus);
    printf(" pAddress:  %s\n",
        incoming_event->data.amSessionStart.pAddress);
    printf(" hSession:  %d\n",
        incoming_event->data.amSessionStart.hSession);

/* Since we will receive two of these events, use a counter to
    keep track of how many we've received. Once we get the second one
    we'll make an call to an EndSession() to end both the sessions. */

    session_start_count++;

    if(session_start_count == 2)
    {
        if(amEndSession(incoming_event-> \
            data.amSessionStart.hSession))
        {
        /* The call returned an ERROR. Abort. */
            continue_processing = 0;
        }
    }
    break;
    case amSessionEndEvent:
    printf("\n-------  RECEIVED EVENT -----\n");
    printf("    event:  %d\n", incoming_event->iEvent);
```

```
            printf("  iStatus:  %d\n",
               incoming_event->data.amSessionEnd.iStatus);
            printf(" hSession:  %d\n",
               incoming_event->data.amSessionEnd.hSession);

            /* Since we will receive two of these events, use a counter to
               keep track of how many we've received. Once we get the second one
               we'll set the continue_processing flag to False. */

            session_end_count++;

            if(session_end_count == 2)
            {
               continue_processing = 0;
            }
            break;
            default:
            printf("\n-------  RECEIVED EVENT -----\n");
            printf("    event:  %d\n", incoming_event->iEvent);
            break;
            }

            /* After "processing" the event, we need to free    the data
               associated with it. To do this, simply pass it
               into the amFreeEvent()function. */

            amFreeEvent(incoming_event);
         }
      }

         if(amShutdown())
         {
         /* This means the call FAILED. WeÌll exit. */
            printf("Call to amShutdown() FAILED\n");
            exit(-1);
         }


      }    /* END of function main() */
```

When the above program is executed, you should see the following output if JetSend core is working properly:

```
>>prog
-------  RECEIVED EVENT -----
```

```
    event:  30
pNetworks:  ip:
 pAddress:  ip:your.local.ip.address#0
-------  RECEIVED EVENT -----
    event:  1
  iStatus:  1
 pAddress:  ip:your.local.ip.address#0
 hSession:  2
-------  RECEIVED EVENT -----
    event:  1
  iStatus:  0
 pAddress:  ip:your.local.ip.address#0
 hSession:  3
-------  RECEIVED EVENT -----
    event:  2
  iStatus:  0
 hSession:  2
-------  RECEIVED EVENT -----
    event:  2
  iStatus:  0
 hSession:  3
>>
```

You may want to reference the tops_device.h file to see what the values for event type and `iStatus` map to. For instance, `amSessionStartEvent` is defined as 1 and `amSessionEndEvent` is defined as 2. Also note that for an `amSessionStartEvent` an `iStatus` of 1 stands for AM_ACTIVE_OK while a 0 stands for AM_PASSIVE_OK. With a successful execution of the above code you should see both a passive and an active `amSessionStartEvent`.

Many more tests can be built based on the above samples of code. However, in order to handle more complicated interactions it makes sense to develop a more modular test harness. You will probably want a separate function that prints the contents of each AMEVENT as it arrives. It would also be helpful for this function to print the "string" representation of each of the non-character event members, such as `iEvent` and `iStatus`.

Basic Test-Harness Helper Functions

In order to develop a more modular test harness, a couple of crucial components needs to be developed. The first of these is an event handler. You can continue in a fashion similar to the above code samples and simply add more case statements to the `iEvent` switch. A different method is to pass the received event into an event handler that calls the appropriate function based on the event type—this is the method we will implement. The second important component is a function that will print out the event when it is received.

The complete code listing of the module called js_test_support.c is given below. The functions it contains will not only allow us to create more complicated tests, but can also be used as the basis of a true JetSend appliance.

js_test_support.c:

```
/*****************************************************************************

 * File:         js_test_support.c
 * Description:  JetSend test-support module
 * Package:      JetSend ADK (TOPS)
 *
 * (c) Copyright 1996, Hewlett-Packard Company, all rights reserved.
 *
 *****************************************************************/
#include <stdio.h>
#include <stdlib.h>

#include "appl.h"
#include "appl_emlib.h"
#include "appl_em.h"
#include "tops.h"
#include "tops_device.h"
#include "js_test_support.h"


/*=================================================================*/
/*                          Local Function ProtoTypes                          */
/*=================================================================*/

static void printEMLevel (Tbyte* em, Tbyte* level, char* pName);
static void printEMvalue (EMvalue value, char* buffer);
static eventTrigger *findEventFunction(AMEVENT *receivedEvent);


/*=================================================================*/
/*                          Global/External Variables                          */
/*=================================================================*/

/*  The following global char arrays are used for printing in "string" form. The
different values of various data-types used by the AMA layer of TOPS. */

Tchar *AMevent_list[32] = \
               {"nullEvent", "amSessionStartEvent",
        "amSessionEndEvent","amErrorEvent",
```

```
            "conRemoteComponentEvent",
            "psInArrivedEvent","psInGoneEvent",
            "psJobEndEvent", "psSurfaceRequestEvent",
            "psSurfaceReleaseEvent",
            "psDescriptionRequestEvent",
            "psContentRequestEvent", "prInRejectedEvent",
            "prJobStartEvent", "prJobEndEvent",
            "prSurfaceEvent", "prDescriptionEvent",
            "prSurfaceReleaseEvent", "prContentEvent",
            "prContentRejectedEvent",
            "ssLocDeviceRequestEvent", "ssLocJobRequestEvent",
            "ssJobStatusReleasedEvent", "srDeviceStatusEvent",
            "srJobStatusEvent", "asSurfaceArrivedEvent",
            "asSurfaceGoneEvent", "asCardSentEvent",
            "arSurfaceRejectedEvent", "arCardReceivedEvent",
               "amNetworkEvent", "amNewGatewayEvent"};

Tchar *dev_status[4] = {"", "STAT_DEV_READY", "STAT_DEV_BUSY", "STAT_DEV_BROKEN"};

Tchar *job_status[6] = {"",  "STAT_JOB_OK", "STAT_JOB_FAIL", "STAT_JOB_PENDING",
                        "STAT_JOB_PROGRESS", "STAT_JOB_PRECENTAGE"};

TqHead *triggerList;     /* This global used to store the event-trigger list */


/*====================================================================*/
/*                       Module Functions */
/*====================================================================*/


/**********************************************************************
*    FUNCTION:  void initializeJSTestSupport()
*
*    DESCRIPTION:
*       This function is called to initialize the js_test_support
*       module. Its basic purpose is to set up the queue which is used
*       to store the event-function triggers.
*
*    PARAMETERS:
*       NONE
*
*    RETURNS:
*       void
```

```
*************************************************************************/
void initializeJSTestSupport()
{
     triggerList = (TqHead *)Tmalloc(sizeof(TqHead));
     QueueModuleInitialize();
     QueueInitialize(triggerList);
}


/*************************************************************************
*    FUNCTION:  void processEvent(AMEVENT *lclEvent)
*
*    DESCRIPTION:
*       The purpose of this function is to direct the received-event
        to its proper trigger-function. The event-loop in your main()
        function should pass to this function all events that it
        receives from TOPS.
 *
 *   PARAMETERS:
 *      AMEVENT  *lclEvent:  The event pointer to be processed.
 *
 *   RETURNS:
 *      void
 *************************************************************************/
void processEvent(AMEVENT *lclEvent)
{
     eventTrigger *triggerToProcess;

     if(lclEvent == NULL)
     {
        printf("\nERROR -- NULL event sent to processEvent()\n");
     }
     else
     {
        /***
           The first thing to do is traverse the trigger list and
           see if the event that came in is triggered...
        ***/

        triggerToProcess = findEventFunction(lclEvent);

       /***
        If we find a triggered event of this type, we need to
```

```
        call the function for this event-type...
        ***/

        if(triggerToProcess != NULL)
        {
        /* Only call the trigger-function if it exists!Note that
           sometimes there may not be one. */

        if(triggerToProcess->funcName)
              (*triggerToProcess->funcName)(lclEvent);
        }
        else
        {
        /***
           this means we didn't find a trigger for the event...
        ***/
           printf("TEST WARNING:   The event received was UNTRIGGERED\n");
        }
     }
}     /* END of function processEvent() */


/*********************************************************************/
*    FUNCTION: void addNewEventFunction(Tu16 eventType,
*              void (*funcToCall)(AMEVENT *))
*
*    DESCRIPTION:
*       This function is called each time you want to add a new event
        trigger. In other words, for each event that your test harness
        supports, you will need to make a call to this function.
        Typically you will make a series of calls to this function in
        the initialization-part of your test-code.
*
*    PARAMETERS:
*       Tu16 eventType: This is the event associated with the trigger-
*       function. The processAMEvent() function will try to match its
        current event with one found in the list of eventTriggers.
        This is the "key" which is used in the match-comparison.
*
*       void(*funcToCall):  This is a pointer to the actual trigger
*       function. Each event type that your test-code supports
*       needs to have an associated function. This function will then
*       be called upon the arrival of each of the events specified by
```

```
*       the eventType.

*       Every one of these trigger-functions has to take the same
*       input-parameters. We've decided to only pass in the AMEVENT
*       we've just received.
*
*    RETURNS:
*       void
*************************************************************************/

void addNewEventFunction(Tu16 eventType, void(*funcToCall)(AMEVENT *))
{
     eventTrigger *newTrigger;

     newTrigger = (eventTrigger *)Tmalloc(sizeof(eventTrigger));
     newTrigger->theEventType = eventType;
     newTrigger->funcName = funcToCall;

     QueueAddToRear(triggerList, (void *)newTrigger);

}    /* END of function addNewEventFunction() */


/*************************************************************************
*    FUNCTION:  static eventTrigger
*       findEventFunction(AMEVENT *receivedEvent)
*
*
*    DESCRIPTION:
*       The purpose of this function is to trace through the list of
*       triggered events and try to find a match for the
*       receivedEvent. In the event that a match is made, the function
*       will return a pointer to that eventTrigger structure.
*
*    PARAMETERS:
*       AMEVENT   *receivedEvent:  This is a pointer to the AMEVENT for
*                                  which a match will attempt to be
                                   made. Actually only the iEvent-
                                   member of the AMEVENT struc-
*                                  ture is used.
*
*    RETURNS:
*       eventTrigger *:  A pointer to the eventTrigger found that
```

```
                        matches the event-type of the AMEVENT passed
                        in.
*
 ***********************************************************************/

static eventTrigger *findEventFunction(AMEVENT *receivedEvent)
{
     eventTrigger *tempTrigger;
     Ts16 keepGoing = 1;

     /***
        The first thing to do here is traverse the trigger list and
        see if there is an event of this type waiting...
     ***/
     QueuePeekAtFront(triggerList, (void **)&tempTrigger);
     while(keepGoing)
     {
        if(tempTrigger == NULL)
     {
        keepGoing = 0;
     }
     else
     {
        if(tempTrigger->theEventType == receivedEvent->iEvent)
        {
           /***
              this means we found a triggered event of the same
              type as was passed in. We can now break from the
              loop.
           ***/
              keepGoing = 0;
           }
           else
           {
           /***
              if we get here that means we haven't found the event
              in the triggered list yet, so keep looking.
           ***/
              QueuePeekAtNext(triggerList, (void *)tempTrigger,
                        (void **)&tempTrigger);
           }
        }
     }
```

```
     return(tempTrigger);
}    /* END of function findEventFunction() */


/*****************************************************************
*    FUNCTION:  void shutdownJSTestSupport()
*
*    DESCRIPTION:
*       The purpose of this function is to simply clean up the global
*       eventTrigger list which was built by making calls to
        addNewEventFunction(). It should only be called when you want
        to "destroy"the current eventTrigger list and shut down the
        test support module.
*
*    PARAMETERS:
*       none
*
*    RETURNS:
*       void
*****************************************************************/
void shutdownJSTestSupport()
{
eventTrigger *tmpTrigger;

     /***
        We're going to simply traverse the trigger list and
        free them all.
     ***/

if( QueueIsEmpty(triggerList) )
{
     return;
}
else
{
     QueueRemoveFront(triggerList, (void **)&tmpTrigger);
     while(tmpTrigger != NULL)
     {
        Tfree(tmpTrigger);
        QueueRemoveFront(triggerList, (void **)&tmpTrigger);
     }
}
```

```
Tfree(triggerList);

}    /* END of function freeSBSEventFunctionList() */


/*********************************************************************
*    FUNCTION:  void printAMEvent(AMEVENT *event)
*
*    DESCRIPTION:
*       This function can be called whenever you want to display the
        contents of an AMEVENT. In the sample test harness it is
        called each time amGetNextEvent() returns an event. This call
        may also make calls to another function called emPrintf().
        This will only happen in the event that the AMEVENT passed in
        has elements containing e-material.
*
*    PARAMETERS:
*       AMEVENT  *event:  This is the event to be printed.
*
*    RETURNS:
*       void
*********************************************************************/
void printAMEvent(AMEVENT *event)
{
     LOCTEXT *tmpLOCTEXT;
     Tchar   *tmpString;

     printf("\n----- RECEIVED EVENT ------\n");
     printf("      event:  %s  (%x)\n", AMevent_list[event->iEvent],
            event);
     switch(event->iEvent)
     {
        case amSessionStartEvent:
            switch (event->data.amSessionStart.iStatus)
        {
            case AM_PASSIVE_OK:
               printf("    iStatus:  AM_PASSIVE_OK\n");
                  break;
            case AM_ACTIVE_OK:
               printf("    iStatus:  AM_ACTIVE_OK\n");
               break;
            case AM_REJECTED:
```

```
            printf("    iStatus:  AM_REJECTED\n");
            break;
        default:
            printf("    iStatus:  AM_FAIL_SESSION\n");
            break;
    }
        printf("   pAddress:  %s\n",
                event->data.amSessionStart.pAddress);
        if(event->data.amSessionStart.pAlias)
            printf("     pAlias:  %s\n",
                event->data.amSessionStart.pAlias);
        printf("   hSession:  %d\n",
                event->data.amSessionStart.hSession);
        break;
    case amSessionEndEvent:
        switch (event->data.amSessionEnd.iStatus)
        {
            case AM_OK:
                printf("    iStatus:  AM_OK\n");
                break;
            default:
                printf("    iStatus:  AM_TIMEOUT\n");
                break;
        }
        printf("   hSession:  %d\n",
            event->data.amSessionEnd.hSession);
        break;
    case conRemoteComponentEvent:
        printf("     hSession:  %d\n",
                event->data.conRemoteComponent.hSession);
        switch(event->data.conRemoteComponent.iComponentId)
        {
            case SELF_LABEL:
            printf(" iComponentId:  SELF_LABEL\n");
            break;
        case SELF_ICON:
            printf(" iComponentId:  SELF_ICON\n");
            break;
        case SELF_ENCODINGS:
            printf(" iComponentId:  SELF_ENCODINGS\n");
            break;
        default:
            printf(" iComponentId:  %d\n",
```

```
                    event->data.conRemoteComponent.iComponentId);
            break;
        }
        if(event->data.conRemoteComponent.pComponent == NULL)
        {
            printf("    pComponent: ==> NULL\n");
            printf("(NOTE:  This component has Been Deleted by its owner)\n");
        }
        else
        {
            switch(event->data.conRemoteComponent.iComponentId)
        {
            case SELF_LABEL:
                printf("   pComponent:  %s\n", \
                    (Tchar *)event->data.conRemoteComponent.pComponent);
                break;
            case SELF_ICON:
                printf("   pComponent:  %x ==>\n",
                          event->data.conRemoteComponent.pComponent);
                 emPrintf(BufDataAddr(event->data.conRemoteComponent.pComponent));
                break;
            case SELF_ENCODINGS:
                printf("   pComponent:  %x ==>\n",
                          event->data.conRemoteComponent.pComponent);
                 emPrintf(BufDataAddr(event->data.conRemoteComponent.pComponent));
                break;
            default:
                printf("   pComponent:  %x ==>\n",
                          event->data.conRemoteComponent.pComponent);
                break;
            }
        }
        break;
    case psInArrivedEvent:
        printf("     hSession:  %d\n",
               event->data.psInArrived.hSession);
        break;
    case psInGoneEvent:
        printf("     hSession:  %d\n",
               event->data.psInGone.hSession);
        break;
    case psJobEndEvent:
        printf("         iJob:  %d\n", event->data.psJobEnd.iJob);
```

```
            break;
        case psSurfaceRequestEvent:
            printf("     nElement:  %s\n",
                   event->data.psSurfaceRequest.nElement);
            printf("   iRequestId:  %d\n",
                   event->data.psSurfaceRequest.iRequestId);
            break;
        case psSurfaceReleasedEvent:
            printf("     hElement:  %d\n",
                   event->data.psSurfaceReleased.hElement);
            break;
        case psDescriptionRequestEvent:
            printf("     hElement:  %d\n",
                   event->data.psDescriptionRequest.hElement);
            printf("    emRequest:  %x\n",
                   event->data.psDescriptionRequest.emRequest);
               emPrintf(BufDataAddr( event->data.psDescriptionRequest.emRequest));
            printf("   iRequestId:  %d\n",
                   event->data.psDescriptionRequest.iRequestId);
            break;
        case psContentRequestEvent:
            printf("     hElement:  %d\n",
                   event->data.psContentRequest.hElement);
            printf("    emRequest:  %x\n",
                   event->data.psContentRequest.emRequest);
            printf("MaxBufferSize:  %d\n",
                   event->data.psContentRequest.iMaxBufferSize);
               emPrintf(BufDataAddr( event->data.psContentRequest.emRequest));
            printf("   iRequestId:  %d\n",
                   event->data.psContentRequest.iRequestId);
            break;
        case prInRejectedEvent:
            printf("     hSession:  %d\n",
                   event->data.prInRejected.hSession);
            break;
        case prJobStartEvent:
            printf("         nDoc:  %s\n", event->data.prJobStart.nDoc);
            printf("         iJob:  %d\n", event->data.prJobStart.iJob);
            printf("         hDoc:  %d\n", event->data.prJobStart.hDoc);
            printf("emDescription:  %x\n",
                   event->data.prJobStart.emDescription);
               emPrintf(BufDataAddr( event->data.prJobStart.emDescription));
            break;
```

```
      case prJobEndEvent:
         printf("         iJob:  %d\n", event->data.prJobEnd.iJob);
         break;
      case prSurfaceEvent:
         printf("   iRequestId:  %d\n",
                event->data.prSurface.iRequestId);
         printf("     hElement:  %d\n",
                event->data.prSurface.hElement);
         printf("emDescription:  %x\n",
                event->data.prSurface.emDescription);
         if (event->data.prSurface.emDescription)
            emPrintf(BufDataAddr( event->data.prSurface.emDescription));
         break;
      case prDescriptionEvent:
         printf("   iRequestId:  %d\n",
            event->data.prDescription.iRequestId);
         printf("emDescription:  %x\n",
                event->data.prDescription.emDescription);
         if (event->data.prDescription.emDescription)
            emPrintf(BufDataAddr( event->data.prDescription.emDescription));
         break;
      case prSurfaceReleasedEvent:
         printf("     hElement:  %d\n",
                event->data.prSurfaceReleased.hElement);
         break;
      case prContentEvent:
         printf("   iRequestId:  %d\n",
                event->data.prContent.iRequestId);
         printf("        pData:  %x\n", event->data.prContent.pData);
            emPrintf(BufDataAddr( event->data.prContent.pData));
         printf("   bLastBlock:  %d\n",
                event->data.prContent.bLastBlock);
         break;
      case prContentRejectedEvent:
         printf("   iRequestId:  %d\n",
                event->data.prContentRejected.iRequestId);
         break;
#if defined(TOPS_LOCALIZE_SS)
      case ssLocDeviceRequestEvent:
         printf("    pLanguage:  %s\n",
                event->data.ssLocDeviceRequest.pLanguage);
         printf("     iRequest:  %d\n",
                event->data.ssLocDeviceRequest.iRequest);
```

```
            break;
        case ssLocJobRequestEvent:
            printf("         iJob:  %d\n",
                   event->data.ssLocJobRequest.iJob);
            printf("    pLanguage:  %s\n",
                   event->data.ssLocJobRequest.pLanguage);
            printf("     iRequest:  %d\n",
                   event->data.ssLocJobRequest.iRequest);
            break;
    #endif
        case ssJobStatusReleasedEvent:
            printf("         iJob:  %d\n",
                   event->data.ssJobStatusReleased.iJob);
            printf(" iComponentId:  %d\n",
                   event->data.ssJobStatusReleased.iComponentId);
            break;
        case srDeviceStatusEvent:
            printf("     hSession:  %d\n",
                   event->data.srDeviceStatus.hSession);
            printf(" iComponentId:  %d\n",
                   event->data.srDeviceStatus.iComponentId);
            if(event->data.srDeviceStatus.iComponentId == STAT_DEV_STRING)
            {
               tmpLOCTEXT = \
                   (LOCTEXT*)(event->data.srDeviceStatus.pComponent);
               if(tmpLOCTEXT == NULL)
            {
               printf("   pComponent:  ==> NULL\n");
            printf("(NOTE:  This device-status component has been released\n");
            }
               else
            {
               printf("   pComponent:  ==>\n");
    #ifdef TOPS_LOCALIZE_SR
               printf("        cLanguage: %s\n", tmpLOCTEXT->cLanguage);
               tmpString = emUnicode2Ascii(tmpLOCTEXT->pString);
               printf("          pString: %s\n", tmpString);
               if(tmpString) Tfree(tmpString);
    #else
                  printf("          pString: %s\n", (Tchar *)tmpLOCTEXT);
    #endif
               }
            }
```

```
            else
         {
            printf("   pComponent:  %s\n",
               dev_status[(Tu32)event->data.srDeviceStatus.pComponent]);
         }
         break;
      case srJobStatusEvent:
         printf("         iJob:  %d\n", event->data.srJobStatus.iJob);
         printf(" iComponentId:  %d\n",
               event->data.srJobStatus.iComponentId);
         if(event->data.srJobStatus.iComponentId == STAT_JOB_STRING)
         {
           tmpLOCTEXT = (LOCTEXT*)(event->data.srJobStatus.pComponent);
           if(tmpLOCTEXT == NULL)
         {
            printf("   pComponent:  ==> NULL\n");
            printf("(NOTE:  This job-status component has been released\n");
         }
            else
         {
             printf("   pComponent:  ==>\n");
#ifdef TOPS_LOCALIZE_SR
            printf("        cLanguage: %s\n", (Tchar *)tmpLOCTEXT->cLanguage);
            tmpString = emUnicode2Ascii(tmpLOCTEXT->pString);
            printf("          pString: %s\n", tmpString);
            if(tmpString) Tfree(tmpString);
#else
            printf("          pString: %s\n", (Tchar *)tmpLOCTEXT);
#endif
         }
         }
         else
         {
            if((Tu32)event->data.srJobStatus.pComponent > 4)
         {
            printf("   pComponent:  STAT_JOB_PERCENTAGE -> %d\n",
      (Tu32)event->data.srJobStatus.pComponent);
         }
            else
         {
            printf("   pComponent:  %s\n",
job_status[(Tu32)event->data.srJobStatus.pComponent]);
         }
```

```
        }
        break;
    case asSurfaceArrivedEvent:
        printf("     hSession:  %d\n",
               event->data.asSurfaceArrived.hSession);
        break;
    case asSurfaceGoneEvent:
        printf("     hSession:  %d\n",
               event->data.asSurfaceGone.hSession);
        break;
    case asCardSentEvent:
        printf("      hTarget:  %d\n",
               event->data.asCardSent.hTarget);
        printf("      iStatus:  %d\n",
               event->data.asCardSent.iStatus);
        if(event->data.asCardSent.pAddress != NULL)
        {
           printf("           pAddress:  %s\n",
                  event->data.asCardSent.pAddress);
        }
        else
        {
           printf("           pAddress:  NULL\n");
        }
        break;
    case arSurfaceRejectedEvent:
        printf("     hSession:  %d\n",
               event->data.arSurfaceRejected.hSession);
        break;
    case arCardReceivedEvent:
        printf("      hSender:  %d\n",
               event->data.arCardReceived.hSender);
        if(event->data.arCardReceived.pData == NULL)
        {
           printf("        pData:  ==> NULL\n");
        }
        else
        {
           printf("        pData:  %x ==>\n",
               event->data.arCardReceived.pData);
           if(event->data.arCardReceived.pData->pAddress != NULL)
        {
           printf("           pAddress:  %s\n",
```

```
event->data.arCardReceived.pData->pAddress);
    }
    else
    {
    printf("           pAddress:  NULL\n");
    }
    if( event->data.arCardReceived.pData->pLabel != NULL)
    {
    printf("             pLabel:  %s\n",
event->data.arCardReceived.pData->pLabel);
    }
       else
    {
       printf("             pLabel:  NULL\n");
    }
       if( event->data.arCardReceived.pData->emIcon != NULL)
    {
       printf("             emIcon:  ==>\n");
       emPrintf(BufDataAddr( event->data.arCardReceived.pData->emIcon));
    }
       else
       {
       printf("             emIcon:  NULL\n");
    }
       if( event->data.arCardReceived.pData->emEncodings != NULL)
    {
       printf("        emEncodings:  ==>\n");
       emPrintf(BufDataAddr( event->data.arCardReceived.pData->emEncodings));
    }
       else
    {
       printf("        emEncodings:  NULL\n");
    }
    }
    break;
case amNetworkEvent:
    if( event->data.amNetwork.pNetworks != NULL)
       printf("  pNetworks:  %s\n",
              event->data.amNetwork.pNetworks);
    if( event->data.amNetwork.pAddress != NULL)
       printf("   pAddress:  %s\n",
              event->data.amNetwork.pAddress);
    break;
```

```
        case amNewGatewayEvent:
            if(event->data.amNewGateway.pAddress != NULL)
                printf("  pAddress:  %s\n",
                       event->data.amNewGateway.pAddress);
            else
                printf("  pAddress:  NULL\n");
            if(event->data.amNewGateway.pReference != NULL)
                printf("  pReference:  %s\n",
                       event->data.amNewGateway.pReference);
            else
                printf("  pReference:  NULL\n");
            printf("      domain:  %d\n",
                   event->data.amNewGateway.domain);
            printf("       local:  %d\n",
                   event->data.amNewGateway.local);
            printf("      status:  %d\n",
                   event->data.amNewGateway.status);
            break;
            default:
                printf("     Unknown Event\n");
            break;
        }
        printf("------------------------------\n");

    }    /* END of function printEvent() */
```

The file js_test_support.h contains the definition of the `eventTrigger` data type as well as the function prototypes for the "public" functions.

In order to illustrate how you can use the js_test_support.c module, we will recreate Test 2 using these functions. The basic algorithm of the main() function should be something like:

```
initialize();
addNewEventFunction();
while(continue)
{
    amGetNextEvent(&event);
    if(event)
    {
        printAMEvent(event)
        processEvent(event);
        amFreeEvent(event)
    }
```

```
    shutdown();
    }
```

The only other important thing you need to provide are the actual "trigger-functions" that will be called by the `processEvent()` function. These functions need to contain code that can process the event data. In the following example, we will only provide functions to handle amNetworkEvents, amSessionStartEvents, and amSessionEndEvents.

The following code listing for Test 3 recreates Test 2 using the js_test_support.c module.

Test 3:

```
#include <stdio.h>
#include <stdlib.h>

#include "appl.h"
#include "tops.h"
#include "tops_device.h"
#include "tops_osp.h"

#include "js_test_support.h"

extern Tu32 giDebugLevel;
extern Tu32 giDebugModule;

int continue_processing = 1;

void amNetworkTrigger(AMEVENT *lclEvent);
void amSessionStartTrigger(AMEVENT *lclEvent);
void amSessionEndTrigger(AMEVENT *lclEvent);

main()
{
    AMEVENT  *incoming_event;

    giDebugLevel = ENTRY|INFO|WARNING|FATAL|FREQUENT;
    giDebugModule = LOG_ALL;


    /* We first need to allow the OSP layer to initialize its
       internal memory/data as well as initialize the TIAPI
       layer. */

    ospSetup();
```

```
        /* Initialize the test_support module */
            initializeJSTestSupport();

        /* Set triggers for the event-types that this test supports */
            addNewEventFunction(amNetworkEvent,  amNetworkTrigger);
            addNewEventFunction(amSessionStartEvent,  amSessionStartTrigger);
            addNewEventFunction(amSessionEndEvent,  amSessionEndTrigger);

        /* The next call will initialize TOPS and give OSP our
            "subaddress" of 0. */

        amInitialize(0);

        while(continue_processing)
        {
            /* We need to call amGetNextEvent() in order for the control-
                thread to be passed on through each layer of TOPS and out
                to the network layer. The function amGetNextEvent() calls
                oipRun() which calls ospRun() which calls xptRun(). In this
                way each layer of TOPS gets a chance to process. */

        amGetNextEvent(&incoming_event);
        if(incoming_event)
        {
            printAMEvent(incoming_event);

            processEvent(incoming_event);

            /* After "processing" the event, we need to free the data
                associated with it. To do this, simply pass it into the
                amFreeEvent() function. */

        amFreeEvent(incoming_event);
        }
    }

    /* Close-down TOPS */
    amShutdown();

    /* Close-down the test-support module */
    shutdownJSTestSupport();

    }     /* END of function main() */
```

```
/*=====================================================================*/
/*                        Event-Trigger Functions */
/*=====================================================================*/


/**********************************************************************/
*    FUNCTION:  void amNetworkTrigger(AMEVENT *lclEvent)
*
*    DESCRIPTION:
*       This function will be called by processEvent() in the event
        that the event being processed is of type amNetworkEvent. In
        this example, the function will first make a call to
        amListenForSession() in order to accept connections over ip.
        It then will make a call to amStartSession() using the local
        ip-address contained in the amNetworkEvent.
*
*    PARAMETERS:
*       AMEVENT  *lclEvent:  This is a pointer to the actual
                             amNetworkEvent.
*
*    RETURNS:
*        void
**********************************************************************/
void amNetworkTrigger(AMEVENT *lclEvent)
{
     if( !strcmp(lclEvent->data.amNetwork.pNetworks, "ip:") )
     {
        if(!amListenForSession(lclEvent->data.amNetwork.pNetworks, 1))
        {
           if(amStartSession(lclEvent->data.amNetwork.pAddress))
           {
              printf("ERROR:  amStartSession Failed\n");
              continue_processing = 0;
           }
        }
        else
        {
           printf("ERROR:  amListenForSession Failed\n");
           continue_processing = 0;
        }
     }
```

```
}


/*********************************************************************/
*    FUNCTION:  void amSessionStartTrigger(AMEVENT *lclEvent)
*
*    DESCRIPTION:
*       This function will be called by processEvent() in case the
*       event being processed is of type amSessionStartEvent. In this
        example, the function will count the number of times it's been
        called. The second time it's called, it will make a call to
        amEndSession() using the session-handle contained within the
        event.
*
*    PARAMETERS:
*    AMEVENT  *lclEvent: This is a pointer to the actual
                         amSessionStartEvent.
*
*    RETURNS:
*       void
*********************************************************************/
void amSessionStartTrigger(AMEVENT *lclEvent)
{
     static int start_count = 0;

     start_count++;

     if(start_count == 2)
     {
        if(amEndSession(lclEvent->data.amSessionStart.hSession))
        {
           printf("ERROR:  amEndSession Failed\n");
           continue_processing = 0;
        }
     }
}


/*********************************************************************/
*    FUNCTION:  void amSessionEndTrigger(AMEVENT *lclEvent)
*
*    DESCRIPTION:
*       This function will be called by processEvent() in case the
```

```
*       event being processed is of type amSessionEndEvent. In this
        example, the function will count the number of times it's been
        called. The second time it's called, it will set the
        continue_processing flag to 0 and thus terminate the event
        processing loop.
*
*    PARAMETERS:
*       AMEVENT  *lclEvent: This is a pointer to the actual
                            amSessionEndEvent.
*
*    RETURNS:
*       void
*********************************************************************/
void amSessionEndTrigger(AMEVENT *lclEvent)
{
     static int end_count = 0;

     end_count++;

     if(end_count == 2)
        continue_processing = 0;
}
```

Notice that in this example the trigger functions do much the same thing as the case statement code in Test 2. In this case, however, the test has a more modular design and can easily be expanded to create hundreds of tests. Upon executing Test 3, you should see the following output:

```
>>prog
----- RECEIVED EVENT ------
      event:  amNetworkEvent  (40017be8)
  pNetworks:  ip:
   pAddress:  ip:your.local.ip.address#0
-------------------------------
----- RECEIVED EVENT ------
      event:  amSessionStartEvent  (40018af8)
    iStatus:  AM_ACTIVE_OK
   pAddress:  ip:your.local.ip.address#0
   hSession:  2
-------------------------------
----- RECEIVED EVENT ------
      event:  amSessionStartEvent  (40018848)
    iStatus:  AM_PASSIVE_OK
   pAddress:  ip:your.local.ip.address#0
   hSession:  3
```

```
------------------------------
----- RECEIVED EVENT ------
      event:  amSessionEndEvent  (40018070)
    iStatus:  AM_OK
   hSession:  2
------------------------------
----- RECEIVED EVENT ------
      event:  amSessionEndEvent  (40018008)
    iStatus:  AM_TIMEOUT
   hSession:  3
------------------------------
>>
```

The last test example further tests the port of JetSend core to your environment. Test 4 will initialize the con, pr, ps, ar, and as activities and then start a session with itself. The act of initializing these activities will cause several different events to be generated by JetSend core. The test will also set a `conRemoteComponent`, causing `conRemoteComponentEvents` to be generated and received.

Test 4:

```
#include <stdio.h>
#include <stdlib.h>

#include "appl.h"
#include "tops.h"
#include "tops_device.h"
#include "tops_osp.h"

#include "js_test_support.h"

extern Tu32 giDebugLevel;
extern Tu32 giDebugModule;

int continue_processing = 1;

void amNetworkTrigger(AMEVENT *lclEvent);
void amSessionEndTrigger(AMEVENT *lclEvent);
void conRemoteComponentTrigger(AMEVENT *lclEvent);

main()
{
     AMEVENT  *incoming_event;
     Tchar    *self_label;
```

```
giDebugLevel = ENTRY|INFO|WARNING|FATAL|FREQUENT;
giDebugModule = LOG_ALL;


/* We first need to allow the OSP layer to initialize its
   internal memory/data as well as initialize the TIAPI
   layer. */

ospSetup();

/* Initialize the test_support module */
   initializeJSTestSupport();

/* Set triggers for the event-types that this test supports.
   Note that only three events have trigger functions. */

addNewEventFunction(amNetworkEvent, amNetworkTrigger);
addNewEventFunction(amSessionStartEvent, NULL);
addNewEventFunction(amSessionEndEvent, amSessionEndTrigger);
addNewEventFunction(conRemoteComponentEvent,
                    conRemoteComponentTrigger);
addNewEventFunction(psInArrivedEvent, NULL);
addNewEventFunction(psInGoneEvent, NULL);
addNewEventFunction(asSurfaceArrivedEvent, NULL);
addNewEventFunction(asSurfaceGoneEvent, NULL);


/* The next series of calls will initialize TOPS and the
   activities this test will use. */

amInitialize(0);
conInitialize(SELF_LABEL);
prInitialize();
psInitialize();
arInitialize(SELF_ADDRESS);
asInitialize();


/* The following calls will set our SELF_LABEL. This will
   cause the generation of conRemoteComponentEvents upon
   making a "connection."  We first have to allocate some
   memory to store the label, and then we copy the label.
```

```
            Note that the TOPS infrastructure will free the self-
            label when it is not needed. */

         self_label = (Tchar *)Tmalloc(sizeof(Tchar) * \
                   (strlen("my self-label")+1));
         if(self_label)
            strcpy(self_label, "my self-label");

         if(conSetComponent(SELF_LABEL, (void *)self_label))
         {
            printf("ERROR:  conSetComponent Failed\n");
            continue_processing = 0;
         }

         while(continue_processing)
         {
         /* We need to call amGetNextEvent() in order for the control-
            thread to be passed on through each layer of TOPS and out
            to the network-layer. The function amGetNextEvent() calls
            oipRun(), which calls ospRun(), which then calls xptRun(). In
            this way each layer of TOPS gets a chance to process. */

            amGetNextEvent(&incoming_event);
            if(incoming_event)
            {
               printAMEvent(incoming_event);

               processEvent(incoming_event);

            /* After "processing" the event, we need to free the data associated with
               it. To do this, simply pass it into the amFreeEvent() function. */

            amFreeEvent(incoming_event);
         }
      }

      /* Close-down TOPS */
      amShutdown();

      /* Close-down the test-support module */
      shutdownJSTestSupport();

      }     /* END of function main() */
```

```
/*====================================================================*/
/*                    Event-Trigger Functions*/
/*====================================================================*/


/**********************************************************************/
*  FUNCTION:  void amNetworkTrigger(AMEVENT *lclEvent)
*
*    DESCRIPTION:
*       This function will be called by processEvent() in case the
*       event being processed is of type amNetworkEvent. In this
        example, the function will first make a call to
        amListenForSession() in order to accept connections over ip.
        It then will make a call to amStartSession() using the local
        ip-address contained in the amNetworkEvent.
*
*    PARAMETERS:
*       AMEVENT  *lclEvent: This is a pointer to the actual
                            amNetworkEvent.
*
*    RETURNS:
*       void
**********************************************************************/
void amNetworkTrigger(AMEVENT *lclEvent)
{

     if( !strcmp(lclEvent->data.amNetwork.pNetworks, "ip:") )
     {
        if(!amListenForSession(lclEvent->data.amNetwork.pNetworks, 1))
        {
           if(amStartSession(lclEvent->data.amNetwork.pAddress))
           {
              printf("ERROR:  amStartSession Failed\n");
              continue_processing = 0;
           }
        }
        else
        {
           printf("ERROR:  amListenForSession Failed\n");
           continue_processing = 0;
        }
```

```
     }
}


/*********************************************************************
*    FUNCTION:  void amSessionEndTrigger(AMEVENT *lclEvent)
*
*    DESCRIPTION:
*       This function will be called by processEvent() in case the
*       event being processed is of type amSessionEndEvent. In this
        example, the function will count the number of times it's been
        called. The second time it's called, it will set the
        continue_processing flag to 0 and thus terminate the event
        processing loop.
*
*    PARAMETERS:
*       AMEVENT  *lclEvent:  This is a pointer to the actual
                             amSessionEndEvent.
*
*    RETURNS:
*       void
*********************************************************************/
void amSessionEndTrigger(AMEVENT *lclEvent)
{
     static int end_count = 0;

     end_count++;

     if(end_count == 2)
        continue_processing = 0;
}


/*********************************************************************
*    FUNCTION:  void conRemoteComponentTrigger(AMEVENT *lclEvent)
*
*    DESCRIPTION:
*       This function will be called by processEvent() in case the
*       event being processed is of type conRemoteComponentEvent. In
        this example, after the second con-event we will make the call
        to end the session.
*
*    PARAMETERS:
```

```
*       AMEVENT  *lclEvent:  This is a pointer to the actual
                             conRemoteComponentEvent.
*
*    RETURNS:
*       void
 ************************************************************************/
void conRemoteComponentTrigger(AMEVENT *lclEvent)
{
     static int con_count = 0;

     con_count++;

     if(con_count == 2)
     {
        if(amEndSession(lclEvent->data.conRemoteComponent.hSession))
        {
           printf("ERROR:  amEndSession Failed\n");
           continue_processing = 0;
        }
     }
}
```

Note that in Test 4 we only created trigger functions for three types of events. The reason for this is that we are simply looking to see if the other expected events are generated. For example, we expect to receive both `psInArrivedEvents` and `asSurfaceArrivedEvents` since we are connecting to a device that has initialized both the pr and ar activities, and yet we are not using any of the data contained within the event.

When the above test is compiled and executed, you should see output very similar to that given below:

```
>>prog
----- RECEIVED EVENT ------
      event:  amNetworkEvent  (4001e940)
  pNetworks:  ip:
   pAddress:  ip:your.local.ip.address#0
-------------------------------
----- RECEIVED EVENT ------
      event:  amSessionStartEvent  (4001f920)
    iStatus:  AM_ACTIVE_OK
   pAddress:  ip:your.local.ip.address#0
   hSession:  7
-------------------------------
----- RECEIVED EVENT ------
      event:  amSessionStartEvent  (4001f938)
```

```
    iStatus:  AM_PASSIVE_OK
   pAddress:  ip:your.local.ip.address#0
   hSession:  8
--------------------------------
----- RECEIVED EVENT ------
     event:  asSurfaceArrivedEvent  (4001f840)
    hSession:  8
--------------------------------
----- RECEIVED EVENT ------
     event:  psInArrivedEvent  (4001fca0)
    hSession:  8
--------------------------------
----- RECEIVED EVENT ------
     event:  asSurfaceArrivedEvent  (4001f5f0)
    hSession:  7
--------------------------------
----- RECEIVED EVENT ------
     event:  psInArrivedEvent  (4001fca0)
    hSession:  7
--------------------------------
----- RECEIVED EVENT ------
     event:  conRemoteComponentEvent  (4001f568)
    hSession:  8
 iComponentId:  SELF_LABEL
   pComponent:  my self-label
--------------------------------
----- RECEIVED EVENT ------
     event:  conRemoteComponentEvent  (4001f498)
    hSession:  7
 iComponentId:  SELF_LABEL
   pComponent:  my self-label
--------------------------------
----- RECEIVED EVENT ------
     event:  asSurfaceGoneEvent  (4001f140)
    hSession:  8
--------------------------------
----- RECEIVED EVENT ------
     event:  psInGoneEvent  (4001f1f0)
    hSession:  8
--------------------------------
----- RECEIVED EVENT ------
     event:  conRemoteComponentEvent  (4001f210)
    hSession:  0
```

```
 iComponentId:  SELF_LABEL
    pComponent: ==> NULL
(NOTE:  This component has Been Deleted by its owner)
-------------------------------
----- RECEIVED EVENT ------
      event:  amSessionEndEvent  (4001f230)
    iStatus:  AM_OK
   hSession:  8
-------------------------------
----- RECEIVED EVENT ------
      event:  asSurfaceGoneEvent  (4001e1e0)
     hSession:  7
-------------------------------
----- RECEIVED EVENT ------
      event:  psInGoneEvent  (4001e940)
     hSession:  7
-------------------------------
----- RECEIVED EVENT ------
      event:  conRemoteComponentEvent  (4001ed78)
     hSession:  0
 iComponentId:  SELF_LABEL
    pComponent: ==> NULL
(NOTE:  This component has Been Deleted by its owner)
-------------------------------
----- RECEIVED EVENT ------
      event:  amSessionEndEvent  (4001ed98)
    iStatus:  AM_OK
   hSession:  7
-------------------------------
>>
```

The four tests described above should provide a simple verification of the JetSend core code on your platform. Of course these tests only cover a fraction of JetSend core's functionality. However, it should be enough to verify that each layer of JetSend core as well as the network-layers are communicating properly.

Verification of JetSend Appliance Negotiation

The second phase of testing your JetSend appliance is the verification of your "device" code and its interaction with JetSend core. This device code interacts between JetSend core and the rest of your appliance subsystem. JetSend core provides the mechanism with which your appliance can exchange the device-independent e-material. It is your responsibility to write the code that acts as the translator between e-material and the data format specific to your appliance. For instance, if you are designing a scanning JetSend appliance, you will need to provide code that can convert

your scanned-image format into the proper e-material representation. Not only that, but you will have to provide the code that performs all of the other maintenance required of a JetSend appliance, such as session-control, negotiation, status, etc.

If you are creating a JetSend appliance that is both a sender and a receiver of e-material, then it will be possible to test your device code against itself. On the other hand, if your appliance is either one or the other, then you will need some sort of JetSend appliance of the opposite type in order to test you code. If you have the opportunity to test your JetSend appliance against itself, be aware that you risk the chance of masking certain defects. Of course, the same is true if you choose to create a generic sender or receiver to test your appliance against, since you will most likely create both appliances using the same understanding of JetSend core and e-material negotiation.

Testing E-Material Negotiation

E-material negotiation is probably the most critical part of your device code. It is within your appliance's creation and parsing of e-material where compatibility problems will most likely be found. JetSend core session-control and even device-status are probably easier to code than is the functionality required for sending and receiving e-material. The reason for this is that while session control and status require a straightforward use of JetSend core event data, sending and receiving e-material requires an interactive negotiation between appliances. This negotiation will be completed successfully only if both appliances conform to the e-material specification provided with this ADK.

When writing your e-material negotiation code, pay special attention to the default encodings listed in the e-material specification. All JetSend appliances must support these encodings during content-negotiation. For example, your JetSend scanning-device must be able to offer its scanned-data in any of the default encodings. So, you will want to exhaustively test the device code responsible for converting the scanned data into e-material.

You must also be prepared to deal with e-material encodings that your device does not support. Your JetSend receiving device may be offered a multitude of optional encodings/attributes of which only a few are supported. Make sure that your device code can parse an e-material description block and select those encodings/attributes that it *can* support.

There may be cases where you are offered a description that does not contain the required default encodings. Even though in this case the other appliance would be at fault, make sure that your device code can handle the situation and cancel the job.

Testing Other Device Code

Apart from e-material negotiation, the other device code you will need to write and test is that which handles the other "activities" supported by your appliance. The most critical device code to test is the am-activity. It is through the am-activity that JetSend core is initialized, sessions are established, and events are given to the appliance. It is through your device code that JetSend core will process data (through calls to `amGetNextEvent()`), listen for and/or actively start sessions, and close sessions. You will want to make sure that your device code properly handles the unexpected ending of sessions. And, if applicable to your appliance, you need to test for failed or rejected session start attempts. Also make sure that your device code properly frees the resources used in the various events returned from JetSend core , either by passing the event to

`amFreeEvent()` or by properly tracking the event-data and freeing it "manually" when it is no longer needed.

Because all other am-activities are optional, your appliance may choose to support as few or as many as make sense. For example, you may choose to only implement the am, con, ps, and sr activities if your JetSend device is a sender of e-material. Listed below are the various am-activities and the areas in them which you may want to pay special attention to during testing.

Contact Activity:

- Make sure your device code properly handles the receiving of NULL `conRemoteComponentEvents`. This can happen when the owner deletes them explicitly through a call to `conSetComponent()` or implicitly when the session is ended.

Push-Send Activity:

- You may receive an unexpected `psInGoneEvent`, which means you can no longer start jobs with that session.
- Make sure your device code can handle those cases where the expected `psInArrivedEvent` does not arrive.

Push-Receive Activity:

- Upon connecting to another device you may receive a `prInRejectedEvent` that needs to be handled properly.
- Remember that a `prEndJobEvent` can occur at anytime after the receipt of a `prJobStartEvent`.
- Note that if the `hElement` member of the `prSurfaceEvent` is zero then the initial surface-request has been rejected by the sender.

Status-Sender Activity:

- If compiling with localization, make sure that your device code can handle any `ssLocDeviceRequestEvents` and `ssLocJobRequestEvents` that you might receive, including those requesting languages your code does not support.
- Be prepared to receive `ssJobStatusReleasedEvents` if your code supports job status.

Status-Receiver Activity:

- Note that the `pComponent` member of both the `srDeviceStatusEvent` and the `srJobStatusEvent` may be NULL.

Address-Sender Activity:

- Be prepared for any unexpected `asSurfaceGoneEvents`. This event will be generated any time a session is closed with a device that supports the address-receiver activity. It can also occur if the address-receiver activity is shut down during a session.

Address-Receiver Activity:

- Your code must handle the occurrence of an `arSurfaceRejectedEvent`. This can happen if a session is ended, the connected device's address sender activity has been shut down, or the connected device does not have the address sender activity running.
- Note that some fields of the `pData` structure contained in the `arCardReceivedEvent` may be NULL.

Verification of JetSend Appliance Operation

The last stage in the JetSend appliance testing sequence tests your appliance as a complete device. This phase might also be referred to as peer testing, since it is through this process that you test your appliance and its performance as a peer interacting with other devices within a system. This type of testing is designed from the customer's point of view. Not only will you need to test the JetSend features found in your appliance, but also any other non-JetSend features your product may support. For instance, you may have simply added JetSend features to a printer that is also to be used as a traditional workstation printer (as in a PC peripheral). In this case your testing would include direct printing from a PC as well as JetSend appliance negotiation that results in a printed page.

Why Do Peer Testing?

There are many reasons why peer testing of your appliance is important. First, JetSend technology may represent a new, untried usage model and functionality that is significantly different from your typical products. The user perspective feedback gained through peer testing can be used to verify the usage model and functionality of your appliance. Second, there are problems that can only be found through peer testing. These problems usually result from the interactions of different components that were not previously tested together. In many cases, development-programs with weak peer testing can have post-release defects that could have been prevented by peer testing. It is likely that in your development program, component testing alone is not sufficient. Lastly, a peer test is the best place to look for cross product problems, or problems that result from interaction between a JetSend device and its environment.

Peer Testing Areas

In order to perform this type of testing with your JetSend appliance you will need access to other JetSend appliances that complement the functionality found in your device. In other words, you cannot test your JetSend scanner without a fully functional JetSend rendering device with which to interact.

Assuming you have access to one or more of these devices, it is recommended that you perform system testing in the following areas.

- Stress Testing

  Stress testing is designed to verify that a device can handle high network traffic and multiple JetSend connections. This type of testing can certify that your JetSend appliance can operate correctly under high levels of network traffic, high levels of one-to-many JetSend transactions, and high levels of many-to-one JetSend transactions.

- Interaction Testing

Interaction testing is designed to look for defects that occur when a device receives both a JetSend and non-JetSend transaction or job. The focus of these tests should be the interaction of the JetSend service with the other services provided by your device. You can use these tests to verify that your JetSend product can operate correctly when put into an environment that contains both JetSend and non-JetSend transactions.

Certification

The last step in testing your product is to run the certification test provided with the JetSend-enabled license.

The certification software will perform a JetSend negotiation and e-material exchange. The software will provide a pass/fail indication as to whether or not your product meets the certification requirements.

Please refer to the certification test tool software and documentation for more information.

Appendix A

JetSend E-Material Specification

The Structure of E-Material

A JetSend **surface** is an object which has a **name**, a **description**, and **content**[1]. A surface is expressed as **e-material**. In simple terms, e-material is comprised of description and content. JetSend uses the description to negotiate the exchange of content. The description enumerates the attributes of the surface; the content is what is normally thought of as the perceivable information.

A successful exchange of a surface between appliances requires that each **attribute level** describing the surface be identified and processed. The process of identifying and processing these levels between the appliances is called **negotiation**. Once the complete surface description has been negotiated, the surface content is exchanged.

The exchange of a surface between two JetSend appliances involves e-material messages in as many as five distinct forms. These are **surface description, description request, description reply, content request,** and **content reply**. An exchange may be initiated by either appliance using a properly formed surface description or description request. A surface description may be completed using description requests and description replies. The surface content exchange is completed using content requests and content replies. Under limited circumstances, surface content may be included in the surface description; this is called **in-line content**. With in-line content and a small description, an exchange may include only a single message. More commonly several messages are exchanged.

E-Material Block

An **e-material block** is represented in a two-column table with **attribute names** in the first column and **attribute values** in the second column. An attribute name is a single specific keyword such as ***vResolution*** or ***vCompression***. An attribute value is one or more values associated with an attribute such as (300,300) pixels per inch or ***vRLE*** (run length encoding). When an attribute value is a list of values, they appear as a space-separated list in the value field.

1. A surface also has properties, a class, and a version. These have no bearing on the e-material specification given here. They are discussed in the JetSend Interaction Protocol Specification.

Attribute	Value
vResolution	(300,300)
vCompression	*vRLE*

The **attribute-value pairs** which appear in the e-material block are applicable to some portion of a JetSend surface. These attribute-value pairs may apply to the whole surface or to a specific portion of the surface. All **attributes** and some **values** are drawn from a limited set of predefined keywords. These are denoted in this specification by a "v" prefix such as ***vEncoding***. Some values are drawn from a set of data types. These are denoted in this specification by an "em" prefix such as *emListType*. In this specification, there are a number of representational types defined to simplify the discussion of the grammar-like structure of JetSend surfaces. These are denoted by italics and are in Hungarian notation, such as *keywordList*.

Certain attributes are associated with a set of values which offer a selection. Each set of values is called a **selection list**. Selection lists appear as a space-separated list in the value column of a two-column table. For example a ***vResolution*** attribute may have a selection list which contains values such as (300,300) pixels per inch or (150,150) pixels per inch. During the negotiation process, each attributes takes one value from the selection list. The selection of each value lends to the identification and processing of each attribute level of the surface description. In this e-material block, the attributes ***vEncoding***, ***vColorSpace***, ***vPixelDepth***, ***vResolution***, and ***vCompression*** take selection lists as values. The attribute ***vSize*** does not.

Attribute	Value
vEncoding	***vImage vText vFile***
vColorSpace	***vGray vSRGB***
vSize	(72000,72000)
vPixelDepth	1 8 24
vResolution	(300,300) (150,150)
vCompression	***vRLE vNone***

As the levels are identified and processed, the attributes which take selection lists and the selected values are enumerated in an e-material block. Attributes which do not take selection lists are omitted. This e-material block is called a **decision block** since it represents the decisions as to the specific attributes of the e-material. The description request and content request include an e-material block which indicates the attributes of the e-material being requested. The table below shows one possible decision block selected from the table above.

Attribute	Value
vEncoding	***vImage***
vColorSpace	***vGray***
vPixelDepth	8
vResolution	(300,300)
vCompression	***vRLE***

An **attribute block** is an e-material block which enumerates all of the attribute-value pairs pertaining to a specific attribute level of the surface description.

Surface Description Block

A **surface description block** is an ordered set of attribute blocks. It may be represented as a three-column table with attribute blocks occupying the last two columns. The first column of the table contains a **decision path** applicable to each attribute block. A decision path is constructed using the values from the decision block. These are concatenated together in a dot notation in the same order as the attribute blocks. Thus an attribute block applicable to an image might be qualified by a decision path such as ***vImage***.***vGray***.8.(300,300).***vRLE*** indicating that it is a run-length encoded image with 300 by 300 dpi resolution and 8 bits per pixel gray scale. This notation describes the composition of the surface which is called the **encoding hierarchy**. The root of this hierarchy is called the **null level**.

Encoding Hierarchy	Attribute	Value
" "	***vEncoding***	***vImage***
vImage	***vSize***	(576000,756000)
vImage	***vColorSpace***	***vGray***
vImage.vGray	***vPixelDepth***	1 8
vImage.vGray.1	***vResolution***	(300,300)
vImage.vGray.1.(300,300)	***vPixels***	(2400,3150)
vImage.vGray.1.(300,300)	***vCompression***	***vRLE***
vImage.vGray.8	***vResolution***	(150,150)
vImage.vGray.8.(150,150)	***vPixels***	(1200,1575)
vImage.vGray.8.(150,150)	***vCompression***	***vRLE***

The encoding hierarchy forms a tree-like structure. Each node in this structure is an attribute level. Each level has an attribute block containing one or more attributes. In each non-terminal level, at least one attribute takes a selection list as a value. The decision path gives the set of decisions down the tree to that level. Each value in the decision path indicates descent to the next level down the encoding hierarchy tree. The last value in the decision path is the name of the attribute level.

A surface description is known to be complete when the last level in the encoding hierarchy is known to be terminal. A level is terminal if none of the attributes it contains take selection lists.

Document Description

A **document** is composed of one or more surfaces. Each surface may be given a name such as "mySurface" or "doc/plane3/image2". A document can be fully described using a four-column document description table with the last three columns occupied by surface description tables. The first column contains the name of the surface being described.

Surface	Encoding Hierarchy	Attribute	Value
"Top"	" "	***vEncoding***	***vPlane***
	vPlane	***vChildFront***	2 ***vSequential*** "A" "B"
	vPlane	***vSize***	(612000,792000)
	vPlane.A	***vPosition***	(18000,18000)
	vPlane.A	***vAttachment***	***vFixed***
	vPlane.B	***vPosition***	(288000,72000)
	vPlane.B	***vAttachment***	***vFloat***
"A"	" "	***vEncoding***	***vImage***
	vImage	***vSize***	(576000,756000)
	vImage	***vColorSpace***	***vGray***
	vImage.vGray	***vPixelDepth***	1 8
	vImage.vGray*.1*	***vResolution***	(300,300)
	vImage.vGray.1.(300,300)	***vPixels***	(2400,3150)
	vImage.vGray.1.(300,300)	***vCompression***	***vRLE***
	vImage.vGray.8	***vResolution***	(150,150)

	vImage.vGray.8.(150,150)	***vPixels***	(1200,1575)
	vImage.vGray.8.(150,150)	***vCompression***	***vRLE***
"B"	" "	***vLanguage***	"en"
	en	***vEncoding***	***vText***
	en.***vText***	***vSize***	(72000,18000)
	en.***vText***	***vSymbolSet***	***vAscii***
	en.***vText***	***vData***	"Hello"

The surfaces in a document are organized in a tree-like structure. A document is composed of pages; the pages are composed of blocks of text or images; and so forth. Certain attributes of a given surface describe the arrangement of **child surfaces** of the level below. These attributes give rise to the **surface hierarchy**. The root of this hierarchy is called the **top-level surface**.

Some surfaces within the surface hierarchy have one or more child surfaces. These child surfaces are organized in one or more ordered **child lists** which are distinct for each **parent surface**. A child list may be well characterized, such as a list of images and text blocks on one side of one page. However the child list may not be fully available at any given time. Consider a stack of pages being scanned by a multi-page scanner. While a number, designation, and content exists for each page, the child list which reflects the composition of the stack is not so readily available in memory at any given time.

The child list when characterized gives the names of the child surfaces related to a parent surface. In the surface description table, this is called a **reference list**.

E-Material Fundamentals

This section gives the formal structure for e-material surface descriptions. The e-material is specified with tables, with the elements of each cell of the table treated in a grammar-like fashion. Strictly speaking there is not an underlying grammar to e-material. These data types would be terminal symbols in a grammar. The higher-level elements are presented here for learning purposes only.

The following notational conventions are used in describing the contents of each cell of the surface description table.

Table A.1: E-Material Symbols

Symbol	Interpretation
::=	may be replaced by
|	An operator, designating an EXCLUSIVE OR, meaning to chose one among the vertical bars.
[ ]	The contents of the brackets are optional
{}	The contents of the braces delineate a set from which choices are made when the operators defined above are used and the scope is ambiguous.
Bold	Any symbol expressed in a bold typeface is an explicit symbol.
Italic	Any symbol expressed in an italic typeface is a predefined keyword, defined for use in this document.
BoldItalic	Any symbol expressed in a bold/italic typeface is an explicit predefined keyword.
null	A decision path of zero length. A reference to the root level of the encoding hierarchy.
.	period character
#	pound character
( )	parenthesis characters
“ ”	quote characters

Data Types

The following are e-material data types.

Table A.2: E-Material Data Types

Assignment	Example
emIntType ::= 32 bit signed integer	10
emIntPairType ::= (emIntType , emIntType)	(100, 200)
emRangeType ::= (emIntType..emIntType)	(100..600)
emRangePairType ::= (emIntType..emIntType, emIntType..emIntType)	(100..600, 200..600)
emStringType ::= null-terminated ASCII string	“string”
emKeyType ::= 16-bit integer representing predefined keywords that identify specific attributes and values.	***vImage***
emListType ::= emKeyType | emIntType | emIntPairType | emRangeType | emRangePairType | emStringType [[emListType]...]	10 (100,200) (100..600) (100..600, 200..600) “string” ***vImage***
emDataType ::= 8-bit binary data	0xdeadbeef

Representational Types

The following e-material definitions are used in this specification for discussion purposes only. While the definitions resemble a grammar, the underlying interpretation parallels the discussion on the structure of the E-Material Byte Format.

Table A.3: Table 3: E-Material Attribute Block Types

Assignment
attribute ::= emKeyType
value ::= emListType | emKeyType
keyword ::= emKeyType

Many values are represented as e-material lists. These lists fall into three categories. General lists are simply values which are best handled together. The number of elements in the list is variable. Selection lists contain individual values which are selected during the negotiation process. Reference lists contain names of surfaces which are linked together to form pages or documents.

Table A.4: E-Material General List Specification Types

Assignment	**Example**
keywordList ::= emKeyType[[emKeyType]...]	***vGray vSRGB vRGB***
intTripleListType ::= emListType containing an emIntType triple	255 255 255
rangePairListType ::= emListType of one or more emRangeType	(100**..**600) (200**..**600) (500..600)

Selection List, Decision Path, Encoding Hierarchy

The *selectionList* can be a large number of token combinations as defined by emListType. Legal combinations are limited by the specific attribute with which the selection list is paired. While individual values in the selection list must match the attribute, there is no practical limit on the number of values in the list. Among the commonly occurring legal combinations are *keySelectionList, stringSelectionList, keyStringSelectionList,* and *intPairRangeSelectionList*.

Table A.5: E-Material Selection List Specification Types

Assignment	Example
selection ::= emKeyType | emIntType | emIntPairType | emStringType	***vImage vText vFile***
selectionList ::= emListType	
keySelectionList ::= emKeyType[[emKeyType]...]	***vPlane vAssociation***
stringSelectionList ::= emStringType[[emStringType]...]	"special1" "special2"
keyStringSelectionList ::= {emKeyType | emStringType}[[{emKeyType | emStringType}]...]	***vPlane*** "special"
rangeSelectionList ::= emRangeType[[emRangeType]...]	(100**..**600**,** 200**..**600) (200**..**1200**,** 200**..**1200)
intPairRangeTypeSelectionList ::= { emIntPairType | emRangeType }[[{ emIntPairType | emRangeType }]...]	(300,300) (100**..**600**,** 200**..**600)
languageSelectionList ::= *stringSelectionList*	"en" "`en-us`" "`en-gb`" "`en-um`" "`fr`" "`ge`" "`it`" "`sp`" "`da`" "`ja`"
encodingSelectionList ::= *keyStringSelectionList*	***vPlane vImage*** "special"

The ***vLanguage*** attribute may be included in any encoding hierarchy if reference to language and country is desired. This is required for the ***vText*** encoding and option for other encodings. The ***vLanguage*** attribute requests a specific selection list. It identifies the languages and optionally the countries for which applicable characteristics are available. The expected format is *languageCode*[-*countryCode*]. A default language value is not specified.

It is a requirement that a language code be identified as a unique value even if the country code may following the language code as a distinct value. It is optional that the country code may be added to a unique language code if the country code would add further clarification to the value. The language and country codes are described in the ISO 639 and ISO 3166 specifications respectively.

An encoding selection list generally only offers one or more emKeyType values such as ***vImage*** or ***vText***. A special encoding may be added by defining an emStringType in the selection list associated with the ***vEncoding*** attribute and using it uniformly. It is considered unique from the keywords defined as emKeyType, so the emKeyType ***vAssociation*** is not the same as the special encoding emStringType "vAssociation".

The order of selections within the selection list indicates the preference of the sending appliance.

A decision path is constructed from the values in a decision block which are in turn taken from selection lists. The encoding hierarchy is expressed in terms of these decision paths. They are used to identify the level of each attribute block.

Table A.6: E-Material Decision Path Specification Types

Assignment	Example
sl*Keyword ::=* emKeyType from a *selectionList.*	***vColorSpace***
sl*Int ::=* emIntType from a *selectionList.*	8
sl*Pair ::=* emIntPairType from a *selectionList.*	(300,300)
sl*String ::=* emStringType from a *selectionList.*	"special"
selectionHierarchy ::= ***sl****Keyword* | ***sl****Int* | ***sl****Pair* | ***sl****String*	(100**..**600) (200**..**600) (500..600)
decisionPath ::= "" | null | *selectionHierarchy*[[**.***selectionHierarchy*]...]	***vImage***.***vGray***.8.(300,300).***vRLE***

The **null level** provides an entry point for the encoding hierarchy of a JetSend surface. At the null level the decision path is null. The null level takes on any of three distinct characters depending on the situation:

- The null level may contain only the ***vEncoding*** attribute. This is the most common type of null level. Most surfaces begin with such a null level.
- The null level may contain only the ***vLanguage*** attribute. The ***vLanguage*** attribute is required for ***vText*** surfaces. It is occasionally seen with other surface encodings as well.
- When a surface description is incomplete, a description request and description reply must be done to complete the exchange of the surface. The description request and description reply together must act on every attribute level to complete the surface description. Those attributes which take selection lists along with the values already selected appear in the decision block of the description request. Those attributes which take selection lists which not been addressed appear in the decision path of the description reply. Since the break in the ordered list of attributes which take selection lists is established dynamically, any level may become the null level in the description reply.

Child List, Reference List, Surface Hierarchy

The surface hierarchy arises from the use of reference lists. A reference list indicates the structure of the child surfaces in the child list which are linked to this parent surface. When a child list is available, the child surfaces of this list correspond directly to the *childCount* number and the child surface names in the reference list. At times, the child list is not available so the reference list indicates an incomplete picture.

The reference list is a simple e-material list, emListType. Most often the list takes the form of *childCount*, the keyword ***vSequential*** or ***vArbitrary***, and a list of child surface names. A reference list may link together elements on a single page (in the case of the ***vPlane*** encoding) or other pages within a document (in the case of the ***vAssociation*** encoding). Each surface is a separately negotiated job element.

A reference list is only associated with the ***vPlane*** or ***vAssociation*** surface encodings. The other encodings (***vImage***, ***vText***, ***vFile***) can only be terminal nodes in the surface hierarchy. A ***vPlane*** encoding may have two (***vChildFront*** for the front and ***vChildBack*** for the back) child surface lists, one (***vChildFront*** or ***vChildBack***) or none. A ***vAssociation*** encoding must have exactly one (***vChild***) child surface list.

Table A.7: E-Material Reference List Specification Types

Assignment	Example
childCount ::= emIntType	1
reference ::= emStringType	"A"
referenceList ::= *reference*[[*reference*]...]	M3 M4 M7
shortReferenceList ::= *childCount* {**`vArbitrary`** | **`vSequential`**} *referenceList* The positive integer *childCount* matches the number of references in *referenceList*.	3 ***vArbitrary*** M3 M4 M7
longReferenceList ::= *childCount* {**`vArbitrary`** | **`vSequential`**} The *childCount* is a large positive value.	207 ***vSequential***
unknownReferenceList ::= *childCount* {**`vArbitrary`** | **`vSequential`**} The *childCount* is a negative value.	-3 ***vSequential***
zeroReferenceList ::= 0	0
anyReferenceList ::= *shortReferenceList* | *longReferenceList* | *unknownReferenceList* | *zeroReferenceList*	
planeReferenceList ::= *referenceList* | **`vFrontFirst`** | **`vFrontLast`** | **`vBackFirst`** | **`vBackLast`**	***vFrontLast***
assocationReferenceList ::= *referenceList* | **`vFirst`** | **`vLast`**	M3 M4 M7

The *childCount* value may be negative, zero, or positive. Small positive values for *childCount* include a corresponding list of references, enumerating the full contents of the child list. Large positive values of *childCount* indicate that the child list is known but the reference list too long to be included. Other *childCount* values indicate the child list is unknown or zero length. In either case no references are included.

The ***vSequential*** and ***vArbitrary*** keywords are used to indicate whether there are any limitations on the order in which the sending appliance can send surfaces. This allows the receiving appliance to fully control the order of surfaces requested. It lets the receiving appliance know, in what order information, both content and indirect impressions, may be requested. Content information about a child may be in-line, or the content information may be requested for one or more children at a time; however the surface description (impression) of a child must be made one impression at a time.

The value ***vArbitrary*** indicates that the sending appliance can access, and send content information, and impress the child surfaces on the receiver in any order that is requested. The value ***vSequential*** means that the children may be requested only in a predetermined order because of limitations of the sending device.

The remainder of the reference list gives names of other surfaces. These names are present if the complete child list is known and relatively short. When references are included the following interpretation applies:

- The reference names must be unique.
- The # (pound sign) is a reserved symbol.
- The references indicate the children that are to be rendered on the plane.
- The order of the children, is the order in which they should be rendered.
- When references are listed then the number of references must match the number of children specified under the *childCount* value. A partial listing of references is not allowed. If all cannot be listed, or the references are unknown, then they must be “discovered” through the mechanisms available in the content request.

It is possible that the content information for some children may be in-line. This is only allowed when a *referenceList* value is included with the ***vChildFront***, ***vChildBack***, or ***vChild*** attribute. For content that is not in-line, a content request is used.

Required Attributes, Default Encodings

To assure that JetSend appliances communicate under all circumstances, specific decision paths must be present. These are called **default encodings**. Sending appliances must be able to produce e-material with the attributes described by these default decision paths. Similarly receiving appliances must be able to interpret e-material with these characteristics.

Default Encodings
vImage.vGray.1.(300,300).***vRLE***

Default encodings are present to assure that appliances will exchange e-material under normal conditions. These encodings are the lowest common characteristics. **Base encodings** are recommended decision paths which permit appliances to exchange e-material with a higher degree of fidelity. Other attributes and values are considered optional.

Base Encodings
vImage.vGray.8.(150,150).***vRLE***
vImage.vGray.8.(150,150).***vNone***
vImage.vSRGB.24.(150,150).***vNone***

The default encodings are decision paths, each element of which comes from a selection list. Associated with each selection list is an attribute. Each of the attributes whose values make up a default encoding must be present in the surface description.

Encoding for vImage Surface

Definition for vImage

The encoding ***vImage*** usually appears at the null level.

A static image is generally thought of as raster information, a bitmap, which when rendered is viewed as a picture. An image may be rendered as dots on a page, even though those dots visually appear as text, tables, pictures, characters, and so forth. An image may also be rendered as pixels on a computer screen. An image may be pixels from a camera, scanner, or any similar device.

Images are generally categorized by color space, pixel depth, resolution, and compression technique. The decision paths within the encoding hierarchy are used to handle this information.

Surface Description for vImage

This table describes the encoding selection of ***vImage***. The attributes and selection values for default encodings are explicitly called out by highlighting. Other attributes and selections appear unhighlighted.

The attributed ***vLanguage*** and ***vEndian*** are optional. This causes the decision path syntax to include optional elements.

Table A.8: Description Block

Encoding Hierarchy	Attribute	Value	Comments
Decision Path**:** null	Optional: ***vLanguage***	***languageSelectionList*** derived from: emListType	The ***vLanguage*** attribute may be included if reference to language and country is desired. A default language value is not specified.
Decision Path**:** [***vLanguage***]	Required: ***vEncoding***	*encodingSelectionList* derived from: emListType Definition: ***vImage*** | Additional: ***vText*** | ***vFile*** | ***vPlane*** | ***vAssociation*** | emStringType	A ***vImage*** is defined as a collection of bits representing a rasterized static image. The image composition is defined by the attributes of this encoding. This section defines the ***vImage*** encoding specifically. The other encodings, ***vText, vFile***, ***vPlane***, ***vAssociation***, and any unique emStringType encodings may also be present.

Table A.8: Description Block

Encoding Hierarchy	Attribute	Value	Comments
Decision Path**:** [***vLanguage***.]***vImage***	Required: ***vSize***	emIntPairType	Designates the size of the image relative to the display surface. The first value is the dimension along the **x**-axis, and the second value is the dimension along the **y**-axis based on 72000 units per inch. The receiving device should know about the display region that it is going to use to render the image, the relative size information, and its own capabilities. This attribute gives the receiving device information that may be used in deciding the best way to receive the image data, knowing the image size and the resources that may be required to render it.
Decision Path: [***vLanguage.***]***vImage***	Required: ***vColorSpace***	*keySelectionList* derived from: emListType Default: ***vGray*** | Base: ***vSRGB*** | Optional: ***vRGB*** | ***vCIELAB*** | ***vCMY*** | ***vYCC*** | ***vYUV*** | ***vXYZ***	Used to offer the color spaces available for the image associated with this surface description.
Decision Path**:** [***vLanguage.***]***vImage***	Optional: ***vOpacity***	emIntType	Designated by an unsigned integer value in the range of 0-255. A value of 0 is interpreted as completely transparent; 255 is interpreted as completely opaque. If ***vOpacity*** is omitted, the default value is 255.
Decision Path**:** [***vLanguage.***]***vImage.sl***ColorSpace	Required: ***vPixelDepth***	*intTripleSelectionList* derived from: emListType	Used to offer the pixel depth encoding for the color space selected. The values available are the number of bits per pixel that identify the color value for that pixel. Beside the default and base values for ***vPixelDepth*** listed below, others may be offered by a device.

Table A.8: Description Block

Encoding Hierarchy	Attribute	Value	Comments
[***vLanguage.***]***vImage. vGray***		Default: 1 Base: 8 Optional: 4	The value 1 only applies when a ***vGray*** color space is selected. It means one bit per pixel. This is a monochrome color space selection. The value 8 means that there are eight bits per pixel used to designate a level of ***vGray***.
[***vLanguage.***]***vImage. vSRGB***		Base: 24 Optional: 8	The value 24 means 24 bits per pixel.
Decision Path**:** [***vLanguage.***]***vImage. sl***ColorSpace**.** ***sl***PixelDepth	Required: ***vResolution***	***intPairRangeSelection List*** derived from: emListType	Offers one or more "real-world" resolutions that can be used for encoding the associated images' pixel data. The *intPairRangeSelectionList* may consist of explicit horizontal resolution in pixels per inch followed by the vertical resolution in pixels per inch. It may also contain emRangePairType for those appliances which offer the ranges of **x** and **y** coordinates rather than only specific values. The range must be specified from the lowest integer value to the highest integer value.
[***vLanguage.***]***vImage. vGray.1***		Default: (300,300) Optional: other specific range	For monochrome, an offering of 300 dpi horizontal and 300 dpi vertical.
[***vLanguage.***]***vImage. vGray***.8		Base: (150,150) Optional: other specific ranges	For grayscale, 8 bits per pixel, an offering of 150 dpi horizontal and 150 dpi vertical.
[***vLanguage.***]***vImage. vSRGB.***24		Base: (150,150) | Optional: other specific ranges	For color, in the various combinations listed, an offering of 150 dpi and 150 dpi.

Table A.8: Description Block

Encoding Hierarchy	Attribute	Value	Comments
Decision Path**:** [***vLanguage.***]***vImage.*** ***sl***ColorSpace**.** ***sl***PixelDepth**.** ***sl***Resolution	Required: ***vPixels***	emIntPairType	Indicates the image width and image height in pixels specified at the resolution indicated by the ***vResolution*** attribute value.
[***vLanguage.***]***vImage.*** ***vGray.***1. (300,300)		(2400,3000) | other specific pairs	For gray or color images the pixel width and pixel height. This is for an 8 inch by 10 inch image.
[***vLanguage.***]***vImage.*** ***vGray***.8. (150,150)		(1200,1500) | other specific pairs	For gray or color images the pixel width and pixel height. This is for an 8 inch by 10 inch image.
[***vLanguage.***]***vImage.*** ***vSRGB.***24. (150,150)		(1200,1500) | other specific pairs	For gray or color images the pixel width and pixel height. This is for an 8 inch by 10 inch image.
Decision Path**:** [***vLanguage.***]***vImage.*** ***sl***ColorSpace**.** ***sl***PixelDepth**.** ***sl***Resolution	Required: ***vCompression***	***keySelectionList*** Base: ***vNone*** | Optional: ***vRLE*** | ***vTIFF*** | ***vG4*** | ***vJPEG*** | ***vLZW***	Represents compression schemes that are available with various color spaces selected. In each case ***vNone*** is an option which means no compression. Compression standards are identified in “Image Compression Standards” on page A-28.
[***vLanguage.***]***vImage.*** ***vGray****.1.*(300,300)		Default: ***vRLE*** Base: ***vNone*** Optional: ***vTIFF*** | ***vG4*** | ***vLZW***	For the monochrome encoding (1 bit per pixel) *vRLE* is the option.
[***vLanguage.***]***vImage.*** ***vGray****.8.(150,150)*		Base: ***vNone*** | Optional: ***vRLE*** | ***vTIFF*** | ***vJPEG*** | ***vLZW***	The grayscale encoding (8 bits per pixel) must support ***vRLE*** and ***vNone***.

Table A.8: Description Block

Encoding Hierarchy	Attribute	Value	Comments
[***vLanguage.***]***vImage.*** ***vSRGB.***24.*(*150, 150)		Base: ***vNone*** | Optional: ***vRLE*** | ***vTIFF*** | ***vJPEG*** | ***vLZW***	The base encoding for color is ***vSRGB*** with a compression of ***vNone***.
Decision Path**:** [***vLanguage.***]***vImage.*** ***sl***ColorSpace**.** ***sl***PixelDepth**.** ***sl***Resolution**.** ***sl***Compression	Optional: ***vEndian***	***vMSB*** | ***vLSB***	The default value specified is ***vMSB*** or big endian. The ***vEndian*** attribute may be included if reference to endian is desired. This attribute defines the order in which bytes are sent. ***vMSB*** = big endian. ***vLSB*** = little endian.
[***vLanguage.***]***vImage.*** ***vGray****.1.*(300,300). ***vRLE***		***vMSB*** | ***vLSB***	For gray or color images the endian format may be either ***vMSB*** or ***vLSB***
[***vLanguage.***]***vImage.*** ***vGray****.1.*(150,150). ***vNone***		***vMSB*** | ***vLSB***	For gray or color images the endian format may be either ***vMSB*** or ***vLSB***
[***vLanguage.***]***vImage.*** ***vSRGB.***24.(150,150). ***vNone***		***vMSB*** | ***vLSB***	For gray or color images the endian format may be either ***vMSB*** or ***vLSB***
Decision Path**:** [***vLanguage.***]***vImage.*** ***sl***ColorSpace**.** ***sl***PixelDepth**.** ***sl***Resolution**.** ***sl***Compression. [***sl***Endian]	Optional: ***vData***	emDataType	Provides in-line content.

Description Request Block for vImage

It is possible that the number of options available for representing a given surface will exceed the available description size. When this occurs the description must be split between the (initial) surface description block and subsequent surface description replies. Alternately a description request may be issues by the receiving appliance to initiate an exchange.

The receiving appliance will know that there is more description available if one or more selections from a selection list are not further described in a subsequent decision path. In extracting the information from the description block, there is no further information below a certain point in the decision tree that has been followed. Upon finding that no further description

exists (and no in-line data is available), a description request is made to the sending appliance. The response to the request would be a continuation of the description block, offering subsequent choices, and data.

A description request is constructed by listing each attribute from the description already received along with the selection from each selection list. The encoding hierarchy is not used. Thus the receiver identifies the selections made by the receiver while processing the surface description.

The structure of a description request and the corresponding description reply are tightly coupled because the decision paths which form the encoding hierarchy of the description reply dependent on the attributes included in the description request.

Table A.9: Description Request Block

Encoding Hierarchy	Attribute	Value	Comments
(Not Used)	Optional: ***vLanguage***	***sl***Language	Include if available.
	Required: ***vEncoding***	***vImage***	Include if available.
	Required: ***vColorSpace***	***sl***ColorSpace	Include if available.
	Required: ***vPixelDepth***	***sl***PixelDepth	Include if available.
	Required: ***vResolution***	***sl***Resolution	Include if available.
	Required: **Compression**	***sl***Compression	Include if available.
	Optional: **Endian**	***sl***Endian	Include if available.

Description Reply Block for vImage

A description reply is used to deliver additional description information that was not included in any previous surface description. A description reply is only sent in response to a description request.

The reply is formed by processing the request, recognizing the selections and therefore the partial decision path, and building a reply based on the hierarchy below the selected level. The decision path which forms the encoding hierarchy is dependent on the attributes included in the description request block. **Every level included in the description request is omitted from the description reply.** Otherwise the structure of the description reply block is identical to the structure of the

original description block. The various selections from the selection lists are shown to be optional but are only so within the constraints of this request/reply pair.

For completeness, all of the attributes are listed here. It is extremely unlikely that ***vLanguage***, ***vEncoding***, or the other early attributes will ever be absent from the initial description (and thus forced into a description reply block).

Table A.10: Description Reply Block

Encoding Hierarchy	Attribute	Value	Comments
null	Optional: ***vLanguage***	*stringSelectionList*	As described earlier.
[***vLanguage***]	Required: ***vEncoding***	*keyStringSelectionList*	As described earlier.
[[***vLanguage.***]***vImage***]	Require): ***vSize***	emIntPairType	As described earlier.
[[***vLanguage.***]***vImage***]	Required: ***vColorSpace***	*keySelectionList*	As described earlier.
[[***vLanguage.***]***vImage***]	Optional: ***vOpacity***	emIntType	As described earlier.
[[[***vLanguage.***]***vImage.***]***sl***ColorSpace]	Required: ***vPixelDepth***	*intTripleListType*	As described earlier.
[[[[***vLanguage.***]***vImage.***] ***sl***ColorSpace.] ***sl***PixelDepth]	Required: ***vResolution***	*intPairRangeSelectionList*	As described earlier.
[[[[[***vLanguage.***]***vImage.***]***sl***ColorSpace.] ***sl***PixelDepth.] ***sl***Resolution]	Required: ***vCompression***	*keySelectionList*	As described earlier.
[[[[[[***vLanguage.***] ***vImage.***]***sl***ColorSpace.] ***sl***PixelDepth.] ***sl***Resolution.] ***sl***Compression]	Optional: ***vEndian***	*keySelectionList*	As described earlier.
[[[[[[[***vLanguage.***] ***vImage.***]***sl***ColorSpace.] ***sl***PixelDepth.] ***sl***Resolution.] ***sl***Compression.] ***sl***Endian]	Optional: ***vData***	emDataType	As described earlier.

Content Request Block for vImage

The content request for ***vImage*** is used to request image content data not included in the surface description.

The receiving appliance will provide the sending appliance with all of the selections that were made from the surface description to indicate to the sending device what selections were made. In addition, the receiving appliance will also provide the sending appliance with the start location to reference the image data. The sending appliance can then return the image data associated with the request. The content reply consists of the number of items sent to the receiver as well as that number of image data items.

The content request is similar in structure to the description request. A content request is constructed of the attributes and values in a manner similar to that of the description request. To this other information is added which allows the receiving appliance to specify which portions of the full content are needed. The encoding hierarchy portion of the table is not used in a content request.

The first section is used to identify the selections made by the receiver while processing the surface description. The purpose is so that any data returned is consistent with the selections. The second section is used to specify the information requested.

The attribute-value pairs in the first of the content requests are defined based on the encoding, the selections made, and the values from the parent description. In some instances there may be no additional attribute-values pairs defined, so the request would be used only to identify the selections made, and to request the data necessary to render, if it is not provided in-line. The attribute-value pairs in first section are defined as part of the content request for each encoding.

Table A.11: Content Request Block

Encoding Hierarchy	Attribute	Value	Comments
(Not Used)	Optional: ***vLanguage***	*sl*Language	Include if available.
	Required: ***vEncoding***	***vImage***	
	Required: ***vColorSpace***	***sl***ColorSpace	
	Required: ***vPixelDepth***	***sl***PixelDepth	
	Required: ***vResolution***	***sl***Resolution	
	Required: **vCompression**	***sl***Compression	

Table A.11: Content Request Block

	Optional: **Endian**	***sl***Endian	Include if available.
	Optional: ***vStartByte***	emIntType	Acts as an index to the first byte of the data wanted. The first byte of data in the image data has index 0. Either the ***vStartByte*** or ***vStartLine*** attribute must be specified.
	Optional: ***vStartLine***	emIntType	Used as a scan line oriented index. The data returned is in scan lines. The index to the first line is 0. Used only if the compression scheme selected is ***vNone***. Either the ***vStartByte*** or ***vStartLine*** attribute must be specified.
	Optional: ***vNumber***	emIntType	Indicates the `number` of bytes (or lines) requested. The sender will send no more than ***vNumber*** bytes (or lines), but it might send less. If the ***vNumber*** attribute is omitted, then the sender may send any number of bytes (or lines).

Content Reply Block for vImage

A content reply is used to deliver the renderable data to the receiving appliance. For small amounts of content data, the data can be sent "in-line". In many instances a content request is made and the content data is returned in response to the content request.

For ***vImage***, a content reply is composed of raw content data. The data is formatted as one or more blocks of data.

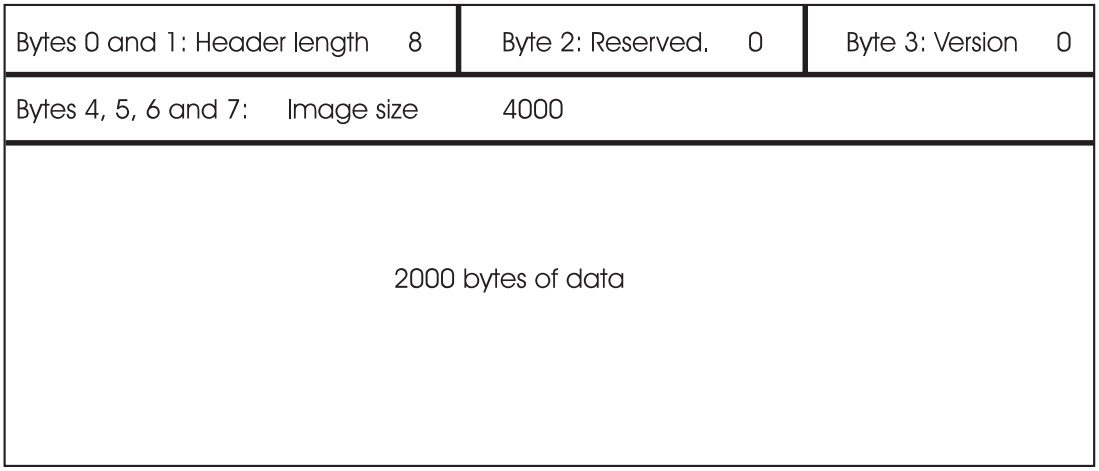
The general description for a content reply of this data format consists of two parts. The first part is a variable length header that exists as part of the first block (only the first block) of content data. The second part is the actual data formatted according to the various applicable attributes. The header information is included in the first block of data for each content request to which the sending device is responding.

The format of the header is as follows:

- Bytes 0 and 1 identify the length of the header.
- Byte 2 identifies the version. The initial version is 0.
- Byte 3 is reserved and initialized to 0
- Bytes 4 - 7 (4 bytes) identify the total number of bytes in the image. This should not be confused with the number of bytes in the block, or the number of bytes that are transferred. This number represents the number of bytes, compressed or uncompressed, for the entire image being represented in the child.
- The header information is in Big Endian. The data portion may be in Little Endian as is indicated in the ***vEndian*** attribute.

Figure A. 1 identifies the general description and has included values to illustrate its use. Note the number of bytes in the image size does not match the number of bytes in the block. This indicates that there is yet another transfer to be made to complete the image.

Figure A. 1 Example of raw content data format



In-Line Content Data for vImage

As seen in both the surface description and description reply block, data may be included with the description. This is advantageous when the in-line content is fairly small. For larger blocks of content data, a separate content block is recommended.

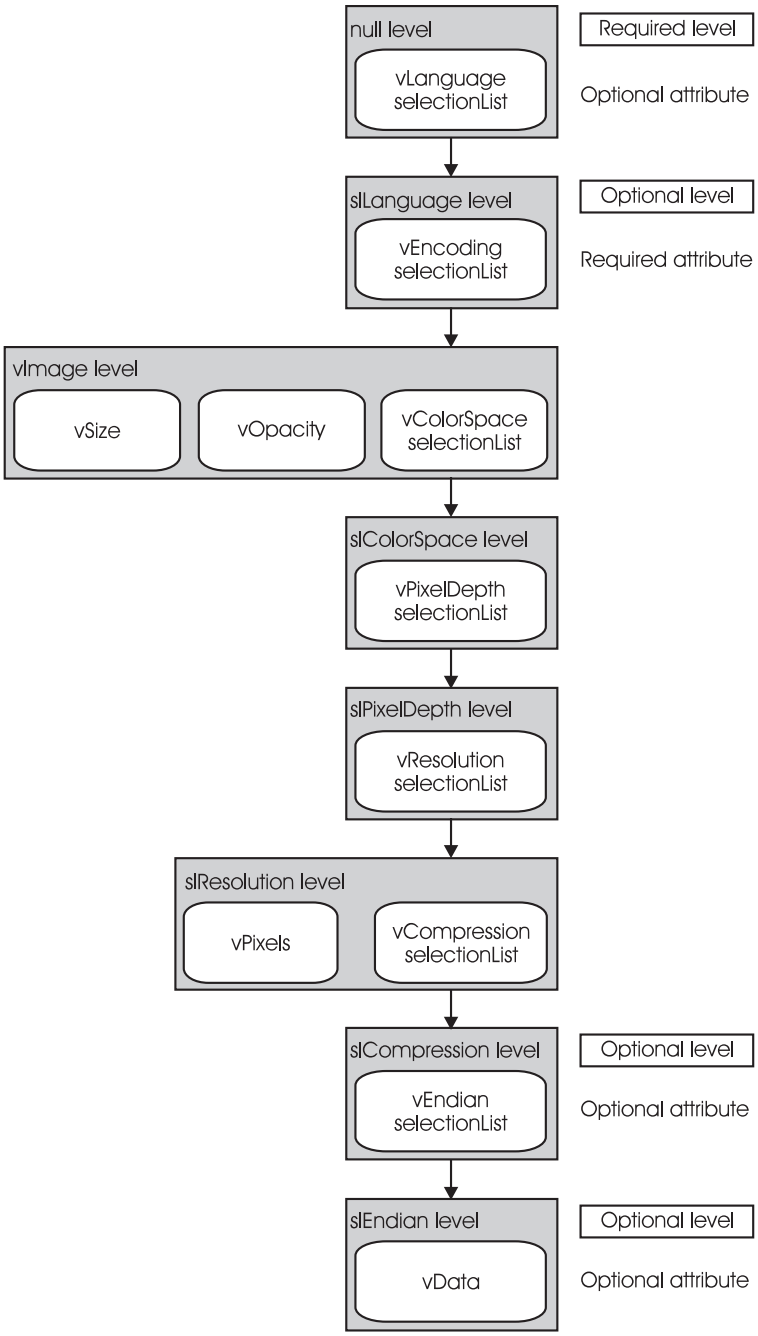
In-line content is used when a small ***vImage*** surface such as an icon is exchanged. A block of binary data is included with the surface description, either the initial description or a subsequent description reply. In-line content must fit completely within the description block. It is not permissible to include partial data in-line.

Guidelines for the E-Material Encoding for vImage

1. The *vLanguage* attribute is an optional attribute where the default value has not been specified. The receiving device may determine not to select a value from the selectionList. In this case the libraries will select the first value described in the level, not in the emListType.
2. For all image encodings the data format, other than that specified in the hierarchy, is word packed, (4 bytes), with each scan line padded to a 4-byte boundary. Compression mechanisms will perform compression on the packed and padded data. This applies to monochrome, gray, and color data.
3. Image data is usually rendered upon some type of media. Please see the guidelines for the ***vPlane*** encoding. This does not mandate that a device must make a ***vImage*** encoding a child of a ***vPlane***, but it is highly recommended that it do so.
4. An image may also live outside of the physical boundaries of a device's medium. In such cases the device must render the data based upon the constraints of the device. In some cases the device may be able to reformat, or resize the image data. Another device may be able to select a different size media. Another device may be constrained to clip the data. The device must determine the most appropriate action for the device and the image.

Hierarchy Diagram for vImage

Note that the ***vLanguage*** attribute and the ***vLanguage*** level are both optional whereas the null level and ***vEncoding*** attribute are both required. If the ***vLanguage*** attribute is omitted so is the ***vLanguage*** level; the ***vEncoding*** attribute then appears in the null level. Similarly the ***vEndian*** attribute and the ***vEndian*** level are optional. If the ***vEndian*** attribute is omitted so is the ***vEndian*** level.



Keywords for Attributes and Values Defined for vImage

Table A.12: Attributes

Attribute	Encoded format
vLanguage	
vEncoding	
vSize	
vOpacity	
vColorSpace	
vPixelDepth	
vResolution	
vPixels	
vCompression	
vEndian	
vStartByte	
vStartLine	
vNumber	
vData	

Table A.13: Values

Value	Encoded format
vEncoding	
vImage	
vText	
vFile	
vPlane	
vAssociation	

Table A.13: Values

vGray	Gray data values range from 0 to n. When the pixel depth is 1 (monochrome) 0 is white and 1 is black. When the pixel depth is greater than 1 (gray) 0 is black and n is white. Data is word packed (4 bytes) and padded to a 4-byte boundary on each scan line.
vSRGB	The data format for SRGB is based on the CCIR 709 specification. A 2.2 gamma is also specified. The data format is n red bits, followed by n green bits, followed by n blue bits where n equals the pixel depth divided by 3 (channels). Data is word packed (4 bytes) and padded to a 4-byte boundary on each scan line.
vRGB	The data format for RGB is n red bits, followed by n green bits, followed by n blue bits, where n equals the pixel depth divided by 3 (channels). Data is word packed (4 bytes) and padded to a 4-byte boundary on each scan line.
vCIELAB	
vCMY	
vYCC	
vYUV	
vXYZ	
vNone	
vRLE	
vTIFF	
vJPEG	
vJPEG-LS	
vG4	
vLZW	
vMSB	
vLSB	

Image Compression Standards

Compression is currently only defined for image data. E-Material descriptors, content reply descriptors or other non-image data will not be compressed or processed by the sending encoders or receiving decoders.

RLE

The E-Material RLE compression method is based on Tagged Image File Format Encoding (TIFF revision 4). This RLE method employs control bytes followed by data bytes. Each control byte in the compressed data sequence is a signed, two's complement byte.

If bit 7 of the control byte is zero (0 <= control byte <= 127) the bytes following are literal. Literal bytes are simply uncompressed data bytes. The number of literal bytes following a control byte is one plus the value of the control byte. Thus, a control byte of 0 means 1 literal byte follows; a control byte of 6 means 7 literal bytes follow; and so on.

If bit 7 of the control byte is 1 (-127 <= control byte <= -1), the byte following the control byte will occur two or more times as decompressed data. A byte following a control byte in this range is called a repeat byte. The control byte's absolute value plus one is the number of times the byte following will occur in the decompressed sequence of bytes. For example, a control byte of-5 means the subsequent byte will occur 6 times as decompressed data.

A control byte of -128 is ignored and is not included in the decompressed data. The byte following a control byte of 128 is treated as the next control byte. It is more efficient to code two consecutive identical bytes as a repeated byte, unless these bytes are preceded and followed by literal bytes. Three-byte repeats should always be encoded using a repeat control byte.

The following is an example of the E-Material RLE compression encoding where tokens within single quotes are single byte character values and the numeric tokens are single byte numbers.

```
Uncompressed Bytes  `I' `I' `I' `I' `I' `I' `S' `E' `E'
Compressed Bytes    -5 `I' 0 `S' -1 `E'
```

JPEG Standard

The E-Material JPEG compression method is based on ISO 10918-1 JPEG Draft International Standard and CCITT Recommendation T.81. JPEG is the Joint Photographic Experts Group working under ISO,CCITT and IEC. The minimum implementation of JPEG shall be baseline sequential DCT with Huffman encoding. Other extended or progressive encodings could be used, but are not guaranteed to be supported by receiving devices. Negotiation of JPEG encoding mechanisms is not supporting within E-Material. Arithmetic encoding should not be used because of patent requirements. Baseline is the base encoding that must be implemented by any JPEG decoder. Since JPEG compression works best on luminance/chrominance color space, the E-Material RGB color data is converted to YCbCr prior to compression. Sampling, quality, compression ratio, and quantization are variable and up to the sending device to determine appropriate parameters. The goal is to obtain high quality images with the smallest possible compressed image size while still having acceptable performance during compression. (Long delays for compression are not acceptable).

JPEG File Interchange Format JFIF specifies an APP0 marker that provides a minimal file header in front of the JPEG compressed data. With this the compressed data may be stored in a file system for different applications to use. The JPEG standard does not specify a file format and recommends that the JFIF APP0 marker not be sent between systems. Senders may use this marker for their internal needs. Receivers should rely on other E-Material specifications for this information and specifically ignore APPn markers.

Application

For E-Material descriptors or content replies that contain image data the first byte of image data is the JPEG compressed data as appropriate, including any and all JPEG markers, frames and scans.

JPEG Implementation

JPEG compression can be based on any source code as long as it adheres to the specification detailed here. One source of JPEG source code is the Independent JPEG Group Jpeg library. It is available form the ftp site ftp.uu.net at graphics/jpeg (Internet IP address 192.48.96.9). This is for reference only. HP does not recommend one vendor over another.

vImage Examples

vImage Example #1

The following is a two inch square image offered only as a monochrome image, no compression or RLE compression, using defaults whenever possible, and none of the data in-line.

	vEncoding	vImage
vImage	vSize	(144000, 144000)
vImage	vColorSpace	vGray
vImage.vGray	vPixelDepth	1
vImage.vGray.1	vResolution	(300,300) (150,150)
vImage.vGray.1.(300,300)	vPixels	(600,600)
vImage.vGray.1.(300,300)	vCompression	vRLE vNone
vImage.vGray.1.(150,150)	vPixels	(300,300)
vImage.vGray.1.(150,150)	vCompression	vNone

vImage Example #2

The following is an example of an image that is offered in a variety of formats. It is offered as a monochrome image, two resolutions, with either no compression or RLE compression. It is also offered as a four-bit grayscale image, in only one resolution, with two possible compression techniques. It is also offered as ***vSRGB*** image with 8 bits per channel (***vSRGB*** has 3 channels which is equal to 24 bits per pixel), in three resolutions and none of those compressed. Of all of the choices offered, only one is in-line, the four bit, gray scale image, with a resolution of 100 x 100, compressed with RLE.

	vEncoding	vImage
vImage	vSize	(72000, 72000)
vImage	vColorSpace	vGray vSRGB
vImage.vGray	vPixelDepth	1 4
vImage.vGray.1	vResolution	(200,100) (300,300)
vImage.vGray.1.(200,100)	vCompression	vNone vRLE
vImage.vGray.1.(300,300)	vCompression	vNone vRLE
vImage.vGray.4	vResolution	(100,100)

vImage.vGray.4.(100,100)	vPixels	(100,100)
vImage.vGray.4.(100,100)	vCompression	vNone vRLE
vImage.vGray.4.(100,100)vRLE	vData	(the data.....)
vImage.vSRGB	vPixelDepth	24
vImage.vSRGB.24	vResolution	(150,150) (300,300) (600,600)
vImage.vSRGB.24.(150,150)	vPixels	(150,150)
vImage.vSRGB.24.(150,150)	vCompression	vNone
vImage.vSRGB.24.(300,300)	vPixels	(300,300)
vImage.vSRGB.24.(300,300)	vCompression	vNone
vImage.vSRGB.24.(600,600)	vPixels	(600,600)
vImage.vSRGB.24.(600,600)	vCompression	vNone

vImage Example #3

The following is an example of an image that is offered as the mandatory encoding and in 24-bit ***vSRGB***. The 24 bit, ***vSRGB*** is offered as JPEG compressed or with no compression, in a fixed default resolution or within a different range. The ***vSRGB***, with a pixel depth of 24 bits, resolution (150,150) and JPEG compressed is offered in-line.

To receiving device ➔

	vEncoding	vImage
vImage	vSize	(72000, 72000)
vImage	vColorSpace	vGray vSRGB
vImage.vGray	vPixelDepth	1
vImage.vGray.1	vResolution	(300,300)
vImage.vSRGB	vPixelDepth	24
vImage.vSRGB.24	vResolution	(150,150) (150..600,150..1200)
vImage.vSRGB.24.(150,150)	vPixels	(150,150)
vImage.vSRGB.24.(150,150)	vCompression	vNone vJPEG
vImage.vSRGB.24.(150,150) . vJPEG	vData	(the data.....)

In this example, the ***vSRGB*** 24-bit, with JPEG is selected. The default resolution is not selected and, a resolution in the range offered is chosen. Offerings within a range cannot be offered as in line data so a Content Request must be made. The amount of data at the resolution requested is unknown so the amount of data requested is not specified. The default is to send as much as possible.

←To sending device

vEncoding	vImage
vColorSpace	vSRGB
vPixelDepth	24
vResolution	(300,600)
vCompression	vJPEG
vStartByte	0

The Content Reply to the request is 5500 bytes of data.

To receiving device ➔

	vData	(the data.....)

It is not known whether or not that is all of the data, so data after the first 5500 bytes is requested through a Content Request.

←To sending device

vEncoding	*vImage*
vColorSpace	*vSRGB*
vPixelDepth	24
vResolution	(300,600)
vCompression	*vJPEG*
vStartByte	5500

The Content Reply is that there is no more data to send and the image can now be rendered.

To receiving device ➔

	vData	

vImage Example #4

The following example is similar to example #3. However only a partial description was available in the initial send. (This shows only the description request and reply. The content request and reply would be identical to example #3.)

To receiving device ➔

	vEncoding	vImage
vImage	vSize	(72000, 72000)
vImage	vColorSpace	vGray vSRGB
vImage.vGray	vPixelDepth	1
vImage.vGray.1	vResolution	(300,300)
vImage.vSRGB	vPixelDepth	24
vImage.vSRGB.24	vResolution	(150,150) (150..600,150..1200)

In this example, there is no information beyond the ***vResolution*** level. A Description Request is created based on the attributes provided along with selections from the selection list. This is directed back to the sending appliance. This includes choosing a resolution in the range.

➔To sending device

vEncoding	vImage
vColorSpace	vSRGB
vPixelDepth	24
vResolution	(300,600)

The Description Reply to the request is the remainder of the description seen in example #4. The rest of the transaction would then follow. Note that the decision paths which appear in the encoding hierarchy are truncated.

To receiving device ➔

	vPixels	(150,150)
	vCompression	vNone vJPEG
vJPEG	vData	(the data.....)

Encoding for vText

Definition for vText

The encoding ***vText*** contains attribute-value pairs associated with textual content. Fundamental attributes include the character set and language associated with the text stream.

The attributes of a text stream represent several levels of elegance. Usually consisting of a character representation (symbol set), and a renderable format (font), text can be visually rendered on a page, screen, or display (plane). Text also has a size which can be described as an area in which it must live. The "size" or area also implies size (height and width) to the text. These fundamental attributes can be used to describe a number of text representations. Other attributes for describing text may be added.

Surface Description for vText

This table describes the layout of a ***vText*** surface description. The attribute-value pairs for the default encoding are explicitly identified by highlighting the default selection. Other attributes and selections appear unhighlighted.

Table A.14: Description Block

Encoding Hierarchy	Attribute	Value	Comments
Decision Path**:** null	Required: ***vLanguage***	*languageSelectionList* derived from: emListType Default: en | Optional: languageCode[-countryCode]	The ***vLanguage*** attribute must be included.
Decision Path**:** ***vLanguage***	Required: ***vEncoding***	*keyStringSelectionList* derived from: emListType Definition: ***vText*** | Additional: ***vImage*** | ***vFile*** | ***vPlane*** ***vAssociation*** | emStringType	A ***vText*** surface is used to communicate written information. The attributes of ***vText*** are defined in this encoding. This section defines the ***vText*** encoding specifically. The other encodings, ***vImage, vFile***, ***vPlane***, ***vAssociation***, and any unique emStringType encodings may also be present.

Table A.14: Description Block

Decision Path**:** ***vLanguage.vText***	Required: ***vSize***	emIntPairType	Designates the size of the text relative to the display surface. The first value is the dimension along the **x**-axis, and the second value is the dimension along the **y**-axis, based on 72000 units per inch. The receiving device should know about the display region that it is going to use to render the data, the relative size information, and its own capabilities. This attribute gives the receiving device information that may be used in deciding the best way to render the text data.
Decision Path**:** ***vLanguage.vText***	Required: ***vSymbolSet***	*keySelectionList* derived from: emListType Default: ***vAscii*** | Optional: ***vUnicode***	Used to determine the appropriate mapping between text values and human readable symbols, or glyphs. The ***vAscii*** symbol set is a mandatory value.This keyword value means that the symbol set to use in interpreting and displaying an eight-bit text value is the default ***vAscii*** symbol set. ***vUnicode*** is also highly recommended.
Decision Path**:** ***vLanguage.vText .vUnicode***	Optional: ***vCodeSet***	*rangePairListType* derived from: emListType	Required when the ***vUnicode*** value is associated with the ***vSymbolSet*** attribute. The attribute ***vCodeSet*** applies only when ***vSymbolSet*** takes on the value of ***vUnicode***. Otherwise it should not be included. It is only optional in this sense. The emRangeType values define the code sets from the Unicode definition that the device supports. The emRangeType values are sequential (lowest to highest) non-overlapping regions. Note that this is not a selection list.
Decision Path**:** ***vLanguage.vText .sl***SymbolSet	Optional: ***vData***	emDataType	Provides in-line content.

Description Request Block for vText

A description request may be issued by the receiving appliance to initiate an exchange. Alternately the description request and description reply may be used in the unlikely event that the ***vText*** surface description exceed the available description size.

A description request is constructed by listing each attribute from the description already received along with the selection from each selection list. The encoding hierarchy is not used. Thus the receiver identifies the selections made by the receiver while processing the surface description.

The structure of a description request and the corresponding description reply are tightly coupled because the decision paths which form the encoding hierarchy of the description reply dependent on the attributes included in the description request.

Table A.15: Description Request Block

Encoding Hierarchy	Attribute	Value	Comments
(Not Used)	Required: ***vLanguage***	***sl***Language	
	Required: ***vEncoding***	***vText***	Include if available.
	Required: ***vSymbolSet***	***sl***SymbolSet	Include if available.

Description Reply Block for vText

A description reply is used to deliver additional description information that was not included in any previous surface description. A description reply is only sent in response to a description request.

The reply is formed by processing the request, recognizing the selections, and therefore the partial decision path, and building a reply based on the hierarchy below the selected level. The decision path which forms the encoding hierarchy is dependent on the attributes included in the description request block. **Every level included in the description request is omitted from the description reply.** Otherwise the structure of the description reply block is identical to the structure of the original description block. The various selections from the selection lists are shown to be optional but are only so within the constraints of this request/reply pair.

For completeness, all of the attributes are listed here. It is extremely unlikely that ***vLanguage***, ***vEncoding***, or the other early attributes will ever be absent from the initial description (and thus forced into a description reply block).

Table A.16: Description Reply Block

Encoding Hierarchy	Attribute	Value	Comments
null	Required: ***vLanguage***	*stringSelectionList*	As described earlier.
vLanguage	Required: ***vEncoding***	*keyStringSelectionList*	As described earlier.
[[***vLanguage.***]***vText***]	Required: ***vSize***	emIntPairType	As described earlier.
[[***vLanguage.***]***vText***]	Required: ***vSymbolSet***	emIntType	As described earlier.
[[***vLanguage.***]***vText***]	Optional: ***vCodeSet***	*rangePairListType*	As described earlier.
[[[***vLanguage.***]***vText.***]***sl***SymbolSet]	Optional: ***vData***	emDataType	As described earlier.

Content Request for vText

The content request for ***vText*** is used to request text content data not included in the surface description.

The receiving appliance will provide the sending appliance with all of the selections that were made from the surface description to indicate to the sending device what selections were made. In addition, the receiving appliance will also provide the sending appliance with the start location to reference the text data. The sending appliance can then return the text data associated with the request. The content reply consists of the number of items sent to the receiver as well as that number of text data items.

The content request is similar to the description request. A content request is constructed of the attributes and values as described with the description request. To this other information is added which allows the receiving appliance to specify which portions of the full content are needed. The hierarchy portion of the table is not used in a content request.

Table A.17: Content Request Block

Encoding Hierarchy	Attribute	Value	Comments
(Not Used)	Required: ***vLanguage***	***sl***Language	

Table A.17: Content Request Block

	Required: ***vEncoding***	***vText***	
	Required: ***vSymbolSet***	***sl***SymbolSet	
	Required: ***vStart***	emIntType	An unsigned integer value acting as an index to the first byte of the data wanted. The first byte of data in the text has index 0.
	Optional: ***vNumber***	emIntType	Indicates the number of bytes requested. The sender will send no more than ***vNumber*** bytes indicated, but it might send less. If ***vNumber*** is omitted the sender can send any number of bytes.

Content Reply Block for vText

A content reply is used to deliver the renderable data to the receiving appliance. For small amounts of content data, the data can be sent "in-line". In many instances a content request is made and the content data is returned in response to the content request. The format of the content reply is based on the content of each encoding type.

For ***vText***, a content reply is composed of raw content data. The data is formatted as one or more blocks of data.

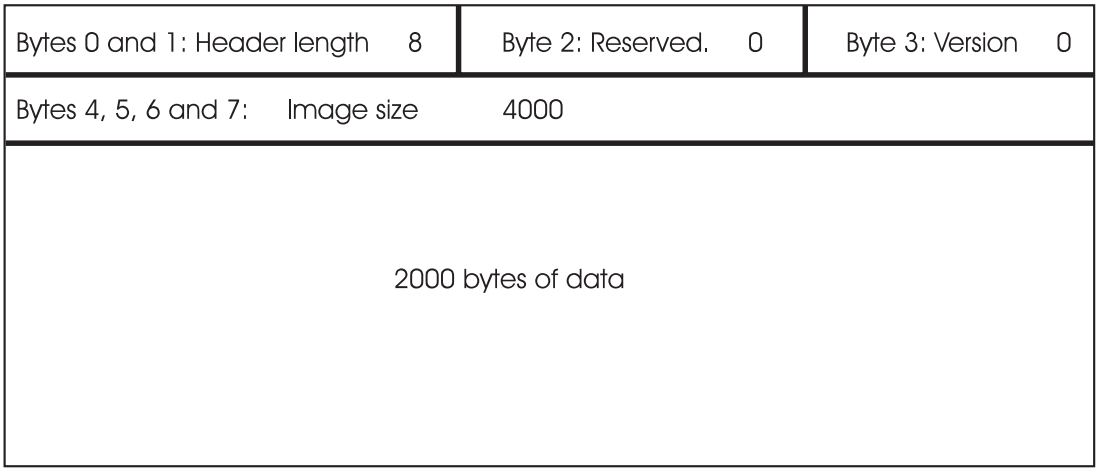
The general description for a content reply of this data format consists of two parts. The first part is a header that exists as part of the first block (only the first block) of content data. The header information is included in the first block of data for each content request to which the sending device is responding.

The format of the header is as follows:

- Bytes 0 and 1 identify the length of the header.
- Byte 2 identifies the version. The initial version is 0.
- Byte 3 is reserved and initialized to 0
- Bytes 4 - 7 (4 bytes) identify the total number of bytes in the text. This should not be confused with the number of bytes in the block, or the number of bytes that are transferred. This number represents the number of bytes, compressed or uncompressed, for the entire text being represented in the child.

Figure A. 2 identifies the general description and has included values to illustrate its use. Note the number of bytes in the text size does not match the number of bytes in the block. This indicates that there is yet another transfer to be made to complete the text.

Figure A. 2 Example of raw content data format



In-Line Content Data for vText

As seen in both the surface description and description reply block, data may be included with the description. This is advantageous when the in-line content is fairly small. For larger blocks of content data, a separate content block is recommended.

In-line content data is most often used with ***vText*** encodings, when a readable message is sent from one appliance to another. A block of binary data is included with the surface description, either the initial description or a subsequent description reply. In-line content must fit completely within the description block. It is not permissible to include partial data in-line.

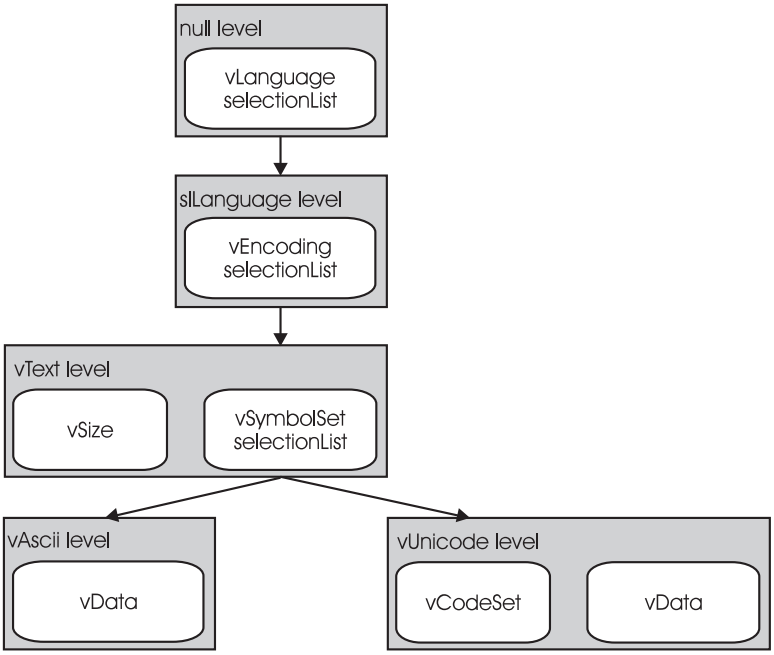
Guidelines for the E-Material Encoding for vText

1. When the ***vText*** value for the ***vEncoding*** attribute is specified, the ***vLanguage*** attribute must also be specified with the "en" value becoming required. Within the ***vText*** encoding is a ***vSymbolSet*** attribute in which the ***vAscii*** symbol set also becomes required. This is to ensure communication between devices. Text and Language are forms of communication that are difficult to specify to ensure a communications mechanism under all circumstances. Not all devices (and the people who view the rendered data) speak the same language. Specifying a specific language and symbol set is the only means of achieving this goal.
2. The emRangeType values associated with the ***vCodeSet*** attribute are used to define the Unicode code sets that a device supports. The devices should specify these ranges as general sets of blocks rather than specific individual character codes. The Unicode code sets are, in general, organized as blocks. There are some associated blocks that are not adjacent and the ***vCodeSet*** attribute does allow for one or more emRangeType. An example: Let's assume (hypothetically) a device supports Basic Latin 0000-007F, Latin 1 Supplement 0080-00FF, Latin Extended-A 0100-017F, Latin Extended-B 0180-024F, a few characters from the General Punctuation block 2002, 2003, 200F, 2012, 2014, a few characters from Superscripts and Subscripts, 2070, 2074, 2084 - 2089, and a few characters from Dingbats 2700-2714, 2723, 2733, 2736, 2754-27BF. To describe these ranges as individual ranges would require a list of 17 ranges.

The general form a device might use to describe this set might be the Latin sets as one range, the General Punctuation and Superscripts/Subscripts as one range, and the Dingbats as one range. All of the characters may not be supported, but to reduce the complexity and mapping of the code sets, this approach may be applied with more efficiency. If further restriction is required, the General Punctuation may be described as a range of 2002 - 2014 and the Superscripts and Subscripts described as a range of 2070 - 2089.

3. Text data is usually rendered upon some type of media. Please see the guidelines for the ***vPlane*** encoding. This does not mandate that a device must make a ***vText*** encoding a child of a vPlane, but it is highly recommended that it do so.

Hierarchy Diagram for vText



Keywords for Attributes and Values Defined for vText

Table A.18: Attributes

Attribute	Encoded format
vLanguage	
vEncoding	
vSize	
vSymbolSet	
vCodeSet	
vStart	
vNumber	
vData	

Table A.19: Values

Value	Encoded format

Table A.19: Values

vEncoding	
vText	
vAscii	
vUnicode	

vText Examples

vText Example #1

The following is an example of simple text, whose data is in-line, using defaults for the optional attributes.

	vLanguage	en
en	vEncoding	vText
en.vText	vSize	(7200, 108000)
en.vText	vSymbolSet	vAscii
en.vText.vAscii	vData	"This is the text."

vText Example #2

The following is an example of simple text, assuming more than one encoding is offered. Both encodings have data in-line.

	vLanguage	en
en	vEncoding	vText
en.vText	vSize	(7200, 108000)
en.vText	vSymbolSet	vAscii vUnicode
en.vText.vAscii	vData	"This is the text."
en.vText.vUnicode	vCodeSet	(0000,007F)
en.vText.vUnicode	vData	"This is the text."

vText Example #3

The following example is very similar to example #2. The only difference is that the vLanguage attribute has been inserted with English (en), United States English (en-us), and French (fr) inserted.

	vLanguage	en en-us fr
en	vEncoding	vText
en.vText	vSize	(7200, 108000)

en.vText	vSymbolSet	vAscii vUnicode
en.vText.vAscii	vData	This is the text.
en.vText.vUnicode	vCodeSet	(0000,007F)
en.vText.vUnicode	vData	“This is the text.”
en-us	vEncoding	vText
en-us.vText	vSize	(7200, 108000)
en-us.vText	vSymbolSet	vAscii vUnicode
en-us.vText.vAscii	vData	“This is the text.”
en-us.vText.vUnicode	vCodeSet	(0000,007F)
en-us.vText.vUnicode	vData	“This is the text”.
fr	vEncoding	vText
fr.vText	vSize	(7200, 108000)
fr.vText	vSymbolSet	vAscii vUnicode
fr.vText.vAscii	vData	“Ceci est le texte”.
fr.vText.vUnicode	vCodeSet	(0000,007F)
fr.vText.vUnicode	vData	“Ceci est le texte.”

Encoding for vFile

The encoding for ***vFile*** is a provision to transfer information when the composition may be unknown by the sender or receiver.

Definition for vFile

The value ***vFile*** indicates a top level encoding, and as defined, provides a mechanism to transfer data as well as any associated icon and naming information.

A file is usually associated with a storage medium and is usually thought of in the abstract format of bits and bytes. In essence, the content of the file may or may not be known. The storage media also applies a naming convention to the file, although it may not provide any significance to the content of the file. Also, in a computerized environment an icon is often associated with the file. Due to the nature of the content of the file, the ***vFile*** encoding may be used as a mechanism to transfer those bits from one device to another while maintaining the relationship of the name, icon and possible content information of the file.

Surface Description for vFile

A ***vFile*** encoding is primarily used for the transfer of binary file type information.

Table A.20: Description Block

Encoding Hierarchy	Attribute	Value	Comments
Decision Path: null	Optional: ***vLanguage***	*languageSelection List* derived from: emListType	The ***vLanguage*** attribute may be included if reference to language and country is desired. A default language value is not specified.
Decision Path: [***vLanguage***]	Required: ***vEncoding***	*keyStringSelection List* Definition: ***vFile*** | Additional: ***vImage*** | ***vText*** | ***vPlane*** | ***vAssociation*** | emStringType	A ***vFile*** is defined as a raw collection of bits which representing any capturable information. Attributes of the file are defined in this encoding. This section defines the ***vFile*** encoding specifically. The other encodings, ***vImage***, ***vText, vPlane***, ***vAssociation,*** and any unique emStringType encodings may also be present.

Table A.20: Description Block

Decision Path**:** [***vLanguage.***]***vFile***	Optional: ***vMimeType***	emStringType	Identifies the characteristics of the ***vFile*** type. The label should adhere to the general guide lines of a mime type construct of: application/msword application/postscript application/zip audio/basic image/tiff etc.
Decision Path**:** [***vLanguage***.]**vFile**	Optional: ***vDataSize***	emIntType	An unsigned integer indicating the size of the binary data.
Decision Path**:** [***vLanguage***.]***vFile***	Optional: ***vChildIcon***	*reference*	Links this file with an icon. The icon is on a separate surface denoted by a reference to that separate child surface. It is often encoded as ***vImage***.
Decision Path**:** [***vLanguage.***]***vFile***	Optional: ***vFilename***	emStringType	Provides a file name as a null terminated text string encoded in the ASCII symbol set.
Decision Path**:** [***vLanguage***.]***vFile***	Optional: ***vData***	emDataType	Provides in-line content.

Description Request Block for vFile

A description request may be issued by the receiving appliance to initiate an exchange. Alternately the description request and description reply may be used in the unlikely event that the ***vFile*** surface description exceed the available description size.

A description request is constructed by listing each attribute from the description already received along with the selection from each selection list. The encoding hierarchy is not used. Thus the receiver identifies the selections made by the receiver while processing the surface description.

The structure of a description request and the corresponding description reply are tightly coupled because the decision paths which form the encoding hierarchy of the description reply dependent on the attributes included in the description request.

Table A.21: Description Request Block

Encoding Hierarchy	Attribute	Value	Comments
(Not Used)	Optional: ***vLanguage***	*sl*Language	Include if available.
	Required: ***vEncoding***	***vFile***	Include if available.

Description Reply Block for vFile

A description reply is used to deliver additional description information that was not included in any previous surface description. A description reply is only sent in response to a description request. In the case of a ***vFile***, the need for a description request and reply is very unlikely.

For completeness, all of the attributes are listed here. It is extremely unlikely that ***vLanguage***, ***vEncoding***, or the other early attributes will ever be absent from the initial description (and thus forced into a description reply block).

Table A.22: Description Reply Block

Encoding Hierarchy	Attribute	Value	Comments
null	Optional: ***vLanguage***	*stringSelectionList*	As described earlier.
[***vLanguage***]	Required: ***vEncoding***	*keyStringSelection List*	As described earlier.
[[***vLanguage.***] ***vFile***]	Optional: ***vMimeType***	emStringType	As described earlier.
[[***vLanguage.***] ***vFile***]	Optional: ***vDataSize***	emIntType	As described earlier.
[[***vLanguage.***] ***vFile***]	Optional: ***vChildIcon***	*reference*	As described earlier.
[[***vLanguage.***] ***vFile***]	Optional: ***vFilename***	emStringType	As described earlier.
[[***vLanguage.***] ***vFile***]	Optional: ***vData***	emDataType	As described earlier.

Content Request for vFile

Use the content request for ***vFile*** to request file content data not included in the surface description. The content request is similar in structure to the description request. A content request is constructed of the attributes and values as described with the description request. To this other information is added which allows the receiving appliance to specify which portions of the full content are needed. The hierarchy portion of the table is not used in a content request.

Table A.23: Content Request Block

Encoding Hierarchy	Attribute	Value	Comments
(Not Used)	Optional: ***vLanguage***	***sl***Language	Include if available.
	Required: ***vEncoding***	***vFile***	Selects the ***vFile*** encoding.
	Required: ***vStart***	emIntType	An unsigned integer value acting as an index to the first byte of the data wanted. The first byte of data in the text has index 0.
	Optional: ***vNumber***	emIntType	An unsigned integer value for the number of bytes requested. The sender will send no more than ***vNumber*** bytes indicated, but it might send less. If ***vNumber*** is omitted the sender can send any number of bytes.

Content Reply Block for vFile

A content reply is used to deliver the renderable data to the receiving appliance. For small amounts of content data, the data can be sent "in-line". In most instances a content request is made and the content data is returned in response to the content request. The format of the content reply is based on the content of each encoding type.

For ***vFile***, a content reply is composed of raw content data. The data is formatted as one or more blocks of data.

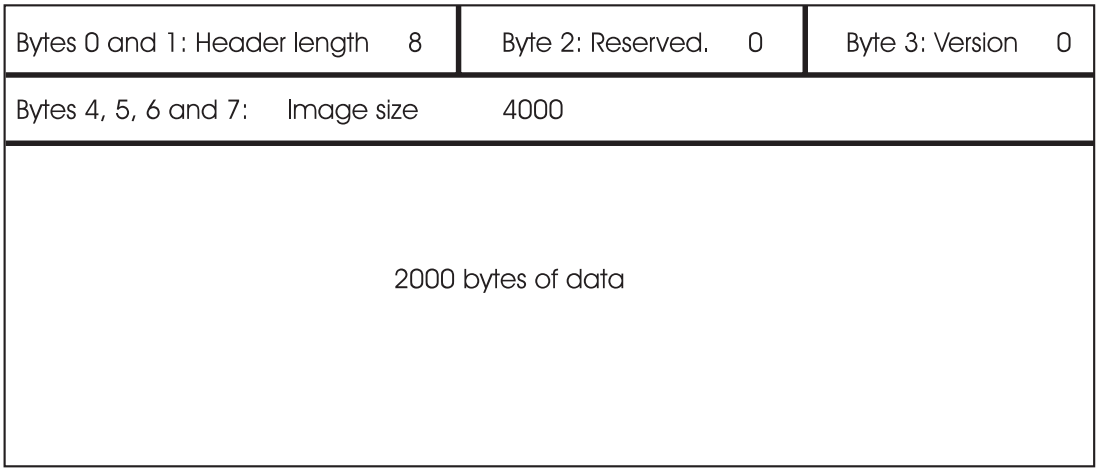
The general description for a content reply of this data format consists of two parts. The first part is a header that exists as part of the first block (only the first block) of content data. The header information is included in the first block of data for each content request to which the sending device is responding.

The format of the header is as follows:

- Bytes 0 and 1 identify the length of the header.
- Byte 2 identifies the version. The initial version is 0.
- Byte 3 is reserved and initialized to 0
- Bytes 4 - 7 (4 bytes) identify the total number of bytes in the image. This should not be confused with the number of bytes in the block, or the number of bytes that are transferred. This number represents the number of bytes, compressed or uncompressed, for the entire image being represented in the child.

Figure A. 3 identifies the general description and has included values to illustrate its use. Note the number of bytes in the size field does not match the number of bytes in the block. This indicates that there is yet another transfer to be made to complete the image.

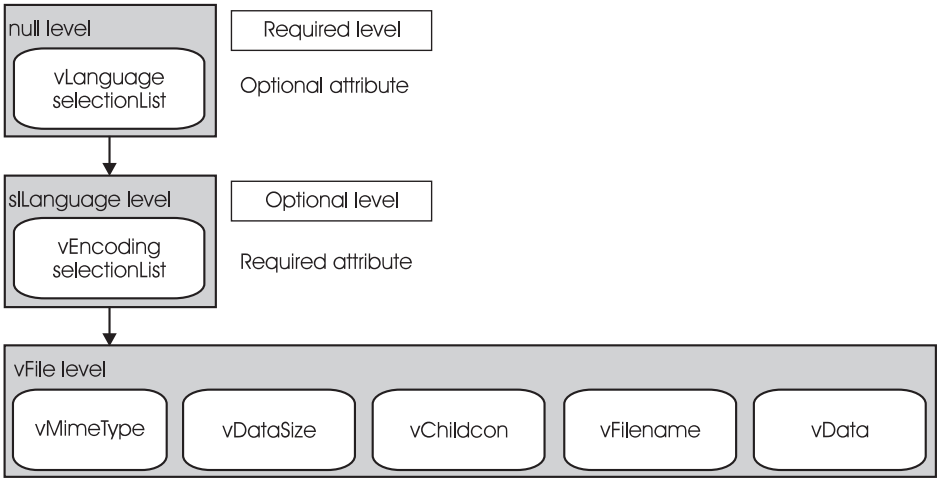
Figure A. 3 Example of raw content data format



Guidelines for the E-Material Encoding for vFile

No specific guidelines for ***vFile*** encoding at present.

Hierarchy Diagram for vFile



Note that the ***vLanguage*** attribute and the ***vLanguage*** level are both optional where as the null level and ***vEncoding*** attribute are both required. If the ***vLanguage*** attribute is omitted so is the ***vLanguage*** level; the ***vEncoding*** attribute then appears in the null level.

Keywords for Attributes and Values Defined for vFile

Table A.24: Attributes

attribute	Encoded format
vLanguage	
vEncoding	
vMimeType	
vDataSize	
vChildIcon	
vStart	
vNumber	
vData	

Table A.25: Values

value	Encoded format
vEncoding	
vFile	
vData	

vFile Examples

vFile Example #1

This is an example of the simplest encoding of a file and nothing else. A file is offered with no icon or label. The amount of data is too large to fit in-line so it must be requested in a series of content requests.

	vEncoding	vFile
vFile	vDataSize	12345543

vFile Example #2

If the file represents a document then it would be possible to offer the encoding as a ***vAssociation*** which represents a document as an alternative to the file (in this case a document with 6 pages)

	vEncoding	vFile vAssociation
vFile	vChildIcon	12
vFile	vDataSize	12345543
vFile	vFilename	"myfile"
vAssociation	vChild	6 vArbitrary 1 2 3 4 5 6

Encoding for vPlane

The ***vPlane*** encoding is primarily designed to represent/describe the following types of information:

- pages
- The relationship between two-dimensional regions on a page. This includes regions that are categorized as images, text, and annotations
- The media upon which to render.

Definition for vPlane

The value ***vPlane*** indicates a top level encoding, and as defined, contains children in order to represent complex information types. It is primarily used to describe the relationship between the children.

In a less abstract form, a plane could represent a page. The page consists of a front side and a back side. The page has size and color. Images and text could represent children which would be rendered on the plane. A plane could also represent a computer screen. The screen layout consists of a single side. The screen has size and color. Images and text could represent multiple children that represent windows with text, sticky notes, and icons.

Surface Description for vPlane

This table describes the layout of a ***vPlane*** surface description.

Table A.26: Description Block

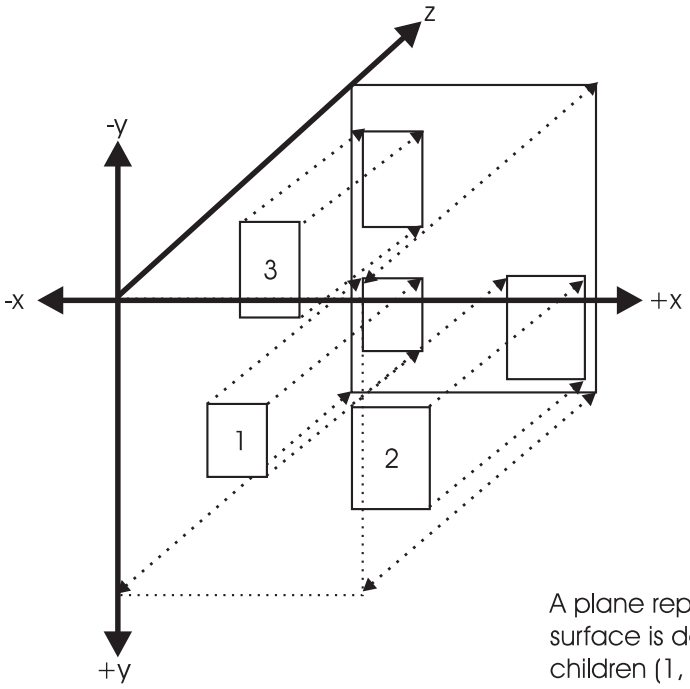
Encoding Hierarchy	Attribute	Value	Comments
Decision Path**:** null	Optional: ***vLanguage***	*languageSelectionList* derived from: emListType	The ***vLanguage*** attribute may be included if reference to language and country is desired. A default language value is not specified.
Decision Path**:** [***vLanguage***]	Required: ***vEncoding***	*encodingSelectionList* derived from: emListType Definition: ***vPlane*** **|** Additional: ***vImage*** | ***vText*** | ***vFile*** | ***vAssociation*** | emStringType	A ***vPlane*** is a page-like surface. It may have two sides. It usually has one or more sub-surfaces as children. The attributes are defined in this encoding. This section defines the ***vPlane*** encoding specifically. The other encodings, ***vImage***, ***vText, vFile***, ***vAssociation,*** and any unique emStringType encodings may also be present.

Encoding Hierarchy	Attribute	Value	Comments
Decision Path**:** [***vLanguage.***]***vPlane***	Required: ***vSize***	emIntPairType	The attribute vSize designates rendering dimensions for the plane being described. Figure A. 5 shows the coordinate system assumed. It is also assumes that the integer values used are in real world coordinates of 1/72000 inch (meaning there are 72000 units per inch) The first integer value is along the **x**-axis, and the second integer value is along the **y**-axis.
Decision Path**:** [***vLanguage.***]***vPlane***	Required: ***vColor***	*intTripleListType*	Indicates the color of the plane. A list of three unsigned integer values, in the range 0-255. The first value is the Red component, the second the Green component, and the third is the Blue component. A 0 value indicates no intensity. 255 indicates full intensity. When the ***vOpacity*** value is 0, ***vColor*** values have no meaning.
Decision Path**:** [***vLanguage.***]***vPlane***	Optional: ***vOpacity***	emIntType	Indicates opacity of the plane to underlying information. Designates by an unsigned integer value in the range of 0-255, a value of 0 being completely transparent, and a value of 255 being completely opaque When ***vOpacity*** is omitted, then the default value is 255.
Decision Path**:** [***vLanguage.***]***vPlane***	Optional: ***vChildFront***	*childCount* {***vArbitrary***| ***vSequential***} [*reference* [*reference* [...]]] otherwise: *anyReferenceList* derived from: emListType	Provides a reference list which links to any child surface on the front of the plane. The *childCount* value indicates the number of known child surfaces. It may be negative, zero, or positive. Small positive values of *childCount* are paired with a matching number of references. Other values are paired with no references. If there are limitations on the order in which the sending appliance can send the surfaces referenced, the ***vSequential*** keyword is used, otherwise the keyword is ***vArbitrary***. This allows the receiving appliance to fully control the order of surfaces requested.

Encoding Hierarchy	Attribute	Value	Comments
Decision Path**:** [***vLanguage.***]***vPlane***	Optional: ***vChildBack***	*childCount* {***vArbitrary*** | ***vSequential***} [*reference* [*reference* [...]]] otherwise: *anyReferenceList* derived from: emListType	Provides a reference list which links to any child surfaces on the back of the plane. The *childCount* value indicates the number of known child surfaces. It may be negative, zero, or positive. Small positive values of *childCount* are paired with a matching number of references. Other values are paired with no references. If there are limitations on the order in which the sending appliance can send the surfaces referenced, the ***vSequential*** keyword is used, otherwise the keyword is ***vArbitrary***. This allows the receiving appliance to fully control the order of surfaces requested.
*Decision Path***:** [***vLanguage.***]***vPlane***	Optional: ***vBackCoord***	emIntType	Specifies the amount of rotation of the back plane coordinate system. The back plane must reflect the dimensions of the front. This attribute is optional if the ***vChildBack*** attribute is omitted. It may also be omitted if there are no changes to the back plane coordinate. If the **x** and **y** values for the ***vSize*** attribute are equal (a square plane) the ***vBackCoord*** values supported are 0, 90, 180, and 270. If the **x** and **y** values for the ***vSize*** attribute are not equal (a rectangular plane) the ***vBackCoord*** values supported are 0 and 180. Any child placed on the plane should be rendered with respect to this coordinate transform. The ***vBackCoord*** is applied as follows: the back plane can be view as if it were flipped about the Temporary **y** axis (reference Figure A. 5 and Figure A. 6). The ***vBackCoord*** can then be applied as a rotation, counter clockwise, around the **x**, **y** intersection (see Figure A. 7). The plane is then moved to the default coordinate position (reference Figure A. 8).

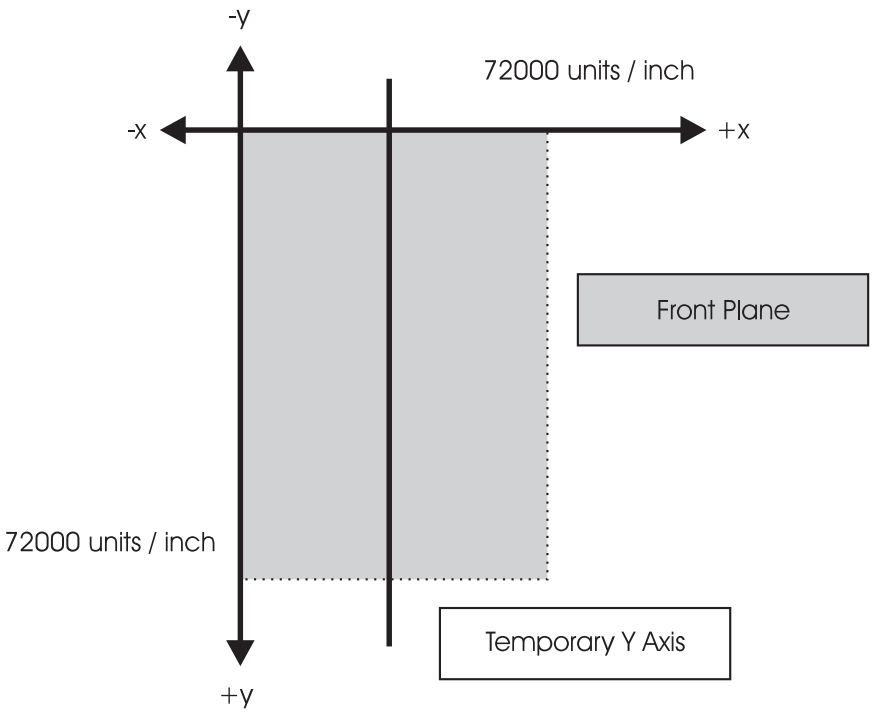
Encoding Hierarchy	Attribute	Value	Comments
Decision Path**:** [***vLanguage***.]***vPlane***.*reference*	Optional: ***vPosition***	emIntPairType	Represents the position of the child with respect to the parent plane. The first value in the pair is the **x**-coordinate, and the second value is the **y**-coordinate. This attribute may be included for the children referenced by the ***vChildFront*** and ***vChildBack*** attributes. This attribute must be listed, uniquely, for each child for which the information is to be provided. This attribute may be provided for any number of the referenced children.
Decision Path**:** [***vLanguage.***]***vPlane***.*reference*	Optional: ***vAttachment***	***vFixed*** | ***vFloating***	Describes how the child may be manipulated relative to the parent intended. The value ***vFixed*** indicates the child should remain fixed at the point specified by the ***vPosition*** attribute. The value ***vFloating*** indicates that the child may be repositioned relative to the parent by appliances capable of such manipulation. The ***vAttachment*** attribute may be included for the children referenced by the ***vChildFront*** and ***vChildBack*** attributes. This attribute must be listed, uniquely, for each child for which the information is to be provided. This attribute may be provided for any number of the referenced children.

Figure A. 4 Graphical view of a vPlane encoding representing a page



A plane representing a surface. The surface is defined and then the children (1, 2, 3) represent images that are placed on the surface. The "z" order of the children determines the order that the images are placed or "rendered" on the surface.

Figure A. 5 The default coordinate system used in E-Material. The Temporary Y Axis is used with the vBackCoord attribute.



The ***vBackCoord*** is applied as follows: the back plane can be viewed as if it were flipped about a Temporary Y axis (reference Figure A.5 and Figure A.6). The ***vBackCoord*** can then be applied as a rotation, counter clockwise, around the x, y intersect (reference Figure A.7) The plane is then moved to the default coordinate position (reference Figure A.8).

Figure A. 6 A back plane flipped about the Y axis.

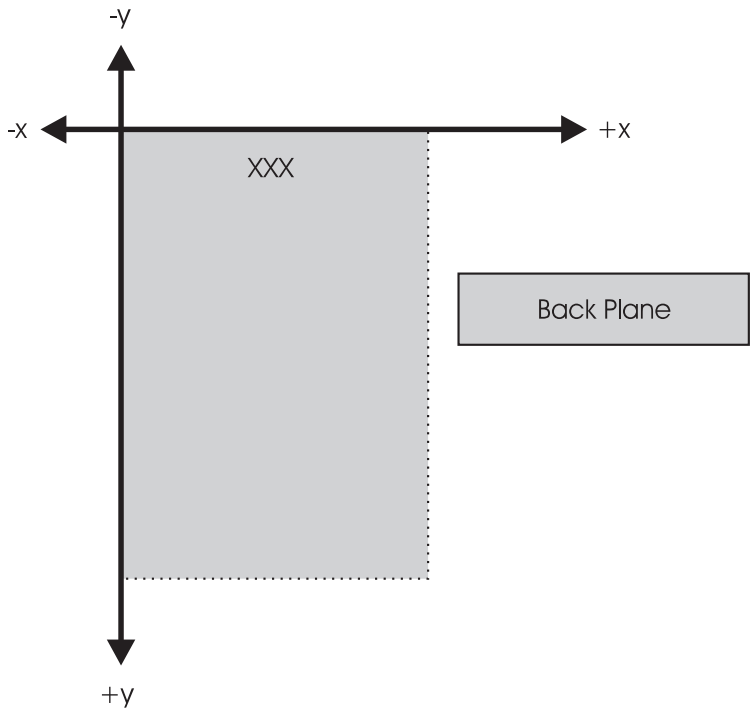


Figure A. 7 A back plane rotated about the X, Y intersection, 180 degrees.

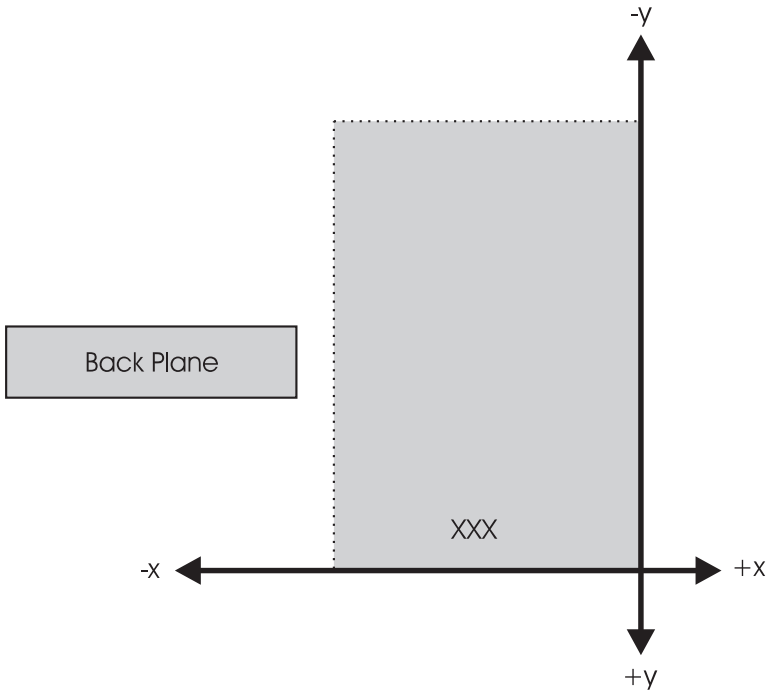
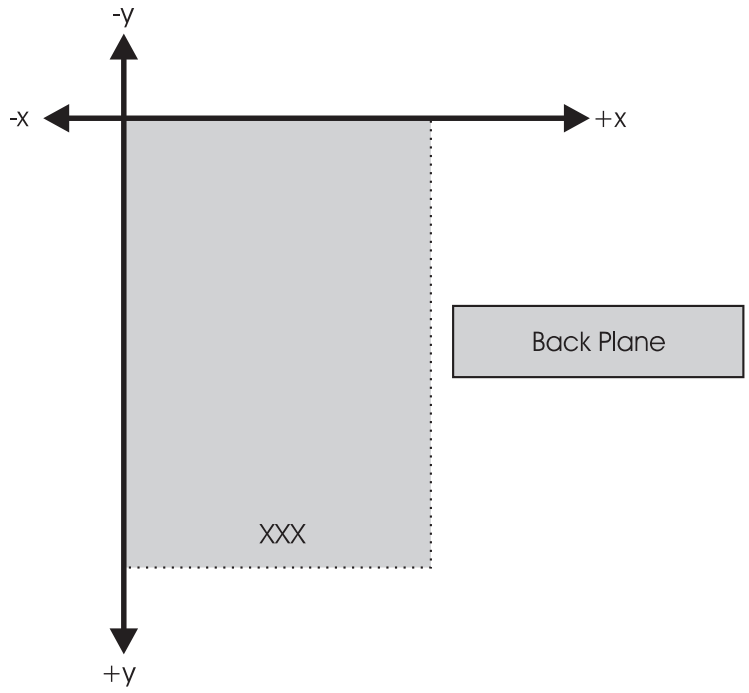


Figure A. 8 A back plane moved to the default coordinate system position



Description Request Block for vPlane

While it is very unlikely that the number of options available for representing a given ***vPlane*** surface would ever exceed the available description size, description request and description reply messages are specified.

A description request is constructed by listing each attribute from the description already received along with the selection from each selection list. The encoding hierarchy is not used. Thus the receiver identifies the selections made by the receiver while processing the surface description.

The structure of a description request and the corresponding description reply are tightly coupled because the decision paths which form the encoding hierarchy of the description reply dependent on the attributes included in the description request.

Table A.27: Description Request Block

Encoding Hierarchy	Attribute	Value	Comments
(Not Used)	Optional: ***vLanguage***	***sl***Language	Include if available.
	Required: ***vEncoding***	***vPlane***	Include if available.

Description Reply Block for vPlane

A description reply is used to deliver additional description information that was not included in any previous surface description. A description reply is only sent in response to a description request.

The reply is formed by processing the request, recognizing the selections and therefore the partial decision path, and building a reply based on the hierarchy below the selected level. The decision path which forms the encoding hierarchy is dependent on the attributes included in the description request block. **Every level included in the description request is omitted from the description reply.** Otherwise the structure of the description reply block is identical to the structure of the original description block. The various selections from the selection lists are shown to be optional but are only so within the constraints of this request/reply pair.

For completeness, all of the attributes are listed here. It is extremely unlikely that ***vLanguage***, ***vEncoding***, or the other early attributes will ever be absent from the initial description (and thus forced into a description reply block).

Table A.28: Description Reply Block

Encoding Hierarchy	Attribute	Value	Comments
null	Optional: ***vLanguage***	*languageSelectionList*	As described earlier.
[***vLanguage***]	Required: ***vEncoding***	*keyStringSelectionList*	As described earlier.
[[***vLanguage.***] ***vPlane***]	Required: ***vSize***	emIntPairType	As described earlier.
[[***vLanguage.***]***vPlane***]	Required: ***vColor***	*intTripleListType*	As described earlier.
[[***vLanguage.***] ***vPlane***]	Optional: ***vOpacity***	emIntType	As described earlier.
[[***vLanguage.***] ***vPlane***]	Optional: ***vChildFront***	*childCount* {***vArbitrary*** | ***vSequential***} [*reference* [*reference* [...]]]	As described earlier.
[[***vLanguage.***] ***vPlane***]	Optional: ***vChildBack***	*childCount* {***vArbitrary*** | ***vSequential***} [*reference* [*reference* [...]]]	As described earlier.

Table A.28: Description Reply Block

Encoding Hierarchy	Attribute	Value	Comments
[[***vLanguage.***] ***vPlane***]	Optional: ***vBackCoord***	emIntType	As described earlier.
[[***vLanguage.***] ***vPlane***.]*reference*	Optional: ***vPosition***	emIntPairType	As described earlier.
[[***vLanguage.***]***vPlane.***]*reference*	Optional: ***vAttachment***	***vFixed*** | ***vFloating***	As described earlier.

Content Request for vPlane

The instances when the receiving device will want to use the content request for ***vPlane*** are:

- To request information about child references when they are not included
- When description information associated with a child reference was not available.

An example when all of the information would not be found in the description block would be when the child name *references* are not available. Those references can be requested through the Content Request using indirect child references (the values of ***vFirst*** and ***vLast***) to indicate that child information is being requested.

A second instance when a Content Request would be used would be to request content information related to a child but relevant to the plane. A child's content information for the encoding for ***vPlane*** consists of ***vPosition*** and ***vAttachment*** attributes and values that that indicate the relationship between the child and the plane.

The receiving device needs a reference to each child. These child references are typically part of the *referenceList* associated with the ***vChildFront*** or ***vChildBack*** attributes or will have already been received through a Description Request as mentioned above. *(Typically the description information for a referenced child is "in-line" due to the few attributes associated with the child at this level)*. When the content information is not in-line a Content Request is used to request the information. The receiving device will provide the sending device with all of the selections that were made from the surface description to indicate to the sending device what selections were made. In addition, the receiving device will also provide the sending device with an explicit list of child references, if available, or an indirect reference (the values of ***vFirst*** and ***vLast***) to indicate a request for child information is being requested. The sending device can then return the description information associated with the child references.

Table A.29: Content Request Block

Encoding Hierarchy	Attribute	Value	Comments
(Not Used)	Optional: ***vLanguage***	***sl***Language	Include if available.
	Required: ***vEncoding***	***vPlane***	Selects the ***vPlane*** encoding.

Table A.29: Content Request Block

	Optional: ***vChild***	{*referenceList* | ***vFrontFirst*** **|*vFrontLast*** **|*vBackFirst*** **|*vBackLast}*** otherwise: *planeReferenceList* derived from: emListType	If a child has previously been referenced using ***vFrontFirst***, ***vFrontLast, vBackFirst***, or ***vBackLast***, these keyword values cannot be used again. If it is a single reference made here, then it specifies a starting point for content information that is to be returned for one or more children. The number requested and the sequence, forward or backward from the reference is specified under the ***vNumber*** attribute. If a *referenceList* is specified, then it is an explicit request for the content of the children specified. The ***vNumber*** attribute is not used. The keywords ***vFrontFirst*** and ***vBackFirst*** are indirect references to the first child in the structure. This applies when the name references are specified as well as when the name references are unknown. The keywords ***vFrontLast*** and ***vBackLast*** are used to refer to the last child in the same way. When, the number of children is unknown or access is sequential, the result of setting the child reference indirectly to the last one may not produce the results expected because the last one may not be accessible. The sending device retains no status of previous requests made, so once ***vFrontFirst, vFrontLast, vBackFirst***, or ***vBackLast*** are used, subsequent requests should use an explicit reference returned in the previous content reply or use the ***vChildNext*** attribute. The ***vChild*** attribute should never be used with the ***vChildNext*** attribute.

Table A.29: Content Request Block

	Optional: ***vChildNext***	*reference*	If a previous child has been referenced then ***vChildNext*** can be used to request content for the next child in the sequence, identified by the sending device. This attribute may not be used if a previous child has not been referenced. This attribute should be omitted if the ***vChild*** attribute is used with a reference name. The reference value is the child name prior to the child information being requested. ***vChildNext*** can be used when the name of the next child is unknown. The number requested and the sequence, forward or backward from the reference is specified under the ***vNumber*** attribute.
	Optional: ***vNumber***	emIntType	This is optional, being used only when a single reference is specified under the ***vChild*** attribute, or when the ***vChild*** attribute is set to ***vFrontFirst, vFrontLast, vBackFirst, vBackLast*** or when the ***vChildNext*** attribute is used. ***vNumber*** is associated with an emIntType value indicating the number of references for which the receiving device is requesting content. When ***vNumber*** is used with a *reference* associated with the ***vChild*** attribute the items returned will include the referenced child. When ***vNumber*** is used with a *reference* associated with the ***vChildNext*** attribute the number of items returned will not include the referenced child but will include children following the reference child. If possible, the sender will send the content information for the children requested, in the order requested. If this attribute is missing, or if the value is zero, then the sender will send the content of as many children as possible from the reference made.

Content Reply for vPlane

A content reply is used to deliver information concerning how child surfaces relate to the parent surface. In the case of ***vPlane,*** this includes the name of the child surfaces in the child lists, the position of each child, and the method of attachment.

The form of the content reply for both ***vPlane*** and ***vAssociation*** is similar to the description reply, in which encoding hierarchy, attributes, and values are returned. This is in contrast to the content reply for ***vImage***, ***vText***, and ***vFile***, in which raw content data is returned.

Table A.30: Content Reply Block

Encoding Hierarchy	Attribute	Value	Comments
Decision Path**:** null	Required: ***vNumber***	emIntType	The number of children in this content reply. If the value is 0 there are no children in this reply. If the value is greater than 0 it indicates the number of children in the reply.
Decision Path**:** null	Optional: ***vNames***	*referenceList* derived from: emListType	This attribute is optional only if the ***vNumber*** of children is 0. The *references* associated with the ***vNames*** attribute are the child references in the order requested. Each child reference must be unique. The # (pound sign) is a reserved symbol.
Decision Path**:** null	Required: ***vRemaining***	emIntType	Specifies the number of children for which a content request has not been made. A negative integer value may be specified indicating the number of children is unknown. A value of 0 indicates that there are no children remaining. A positive integer indicates the absolute number of children remaining.
			For each of the references returned above, the following information should be included for each.
Decision Path**:** null.*reference*	Required: ***vPosition***	emIntPairType	Represents the position of the child with respect to the parent plane. The first value in the pair is the X coordinate, and the second value is the Y coordinate.

Table A.30: Content Reply Block

Decision Path: null.*reference*	Required: ***vAttachment***	***vFixed*** ***vFloating***	Describes how the child is related to the parent and what type of manipulation is intended. The value ***vFixed*** indicates the child should remain "fixed" at the point specified by the ***vPosition*** attribute. If it is "***vFloating***" the ***vPosition*** indicates the point the child should be positioned relative to the parent, but the child could be moved by the user if the rendering device allows such treatment of a child.

Guidelines for the E-Material Encoding for vPlane

1. The ***vPlane*** encoding, in many cases, could be viewed as the media upon which images and text are rendered. Pages, computer displays, status displays, white boards, projection media, etc. are examples of that type of media. A renderable data encoding, such as image or text, should be applied to a media, ***vPlane***. An explicit description, (but not the rule) would be that ***vImage*** and ***vText*** should be children of ***vPlane***.
2. Each reference associated with the ***vChildFront***, ***vChildBack***, ***vChild***, ***vChildNext***, or ***vNames*** attributes must be unique. A mechanism has been provided in which a single surface may be referenced multiple times. If the receiving device wishes to implement this mechanism the receiving device would have the option of requesting the surface multiple times, or maintaining the reference data internally. A "#reference" (pound sign)(reference) may be included when the surface information is used multiple times but the child content (vPosition, vAttachment) information is different. The "#reference" acts as an index to uniquely identify the child and child content. For example: If the surface name is logo, but the "logo" is to be rendered on the page multiple times, the sender could provide the child reference names as logo#1, logo#2, logo#3. The receiving device then has the option of requesting the content information for the position and attachment as logo#1, logo#2 and logo#3 and requesting the surface "logo#1 or logo#2 or logo#3" one time OR the receiving device could request the content information as logo#1, logo#2 and logo#3 and the surfaces as logo#1, logo#2 and logo#3 each as a unique surface.
3. When the *referenceList* does not contain the names of the child surfaces, they must be obtained through inquiry.
4. When the value ***vSequential*** appears, the children may be requested only in a predetermined order because of limitations of the sending device. This implies a processing sequence of:
   - Request content information (which describes the attachment) of next child (using a content request and reply pair on the parent surface)
   - Request the surface description / impression of that child (using a description request and reply pair on that child surface)
   - Request the content of that child (using a content request and reply pair on that child surface)

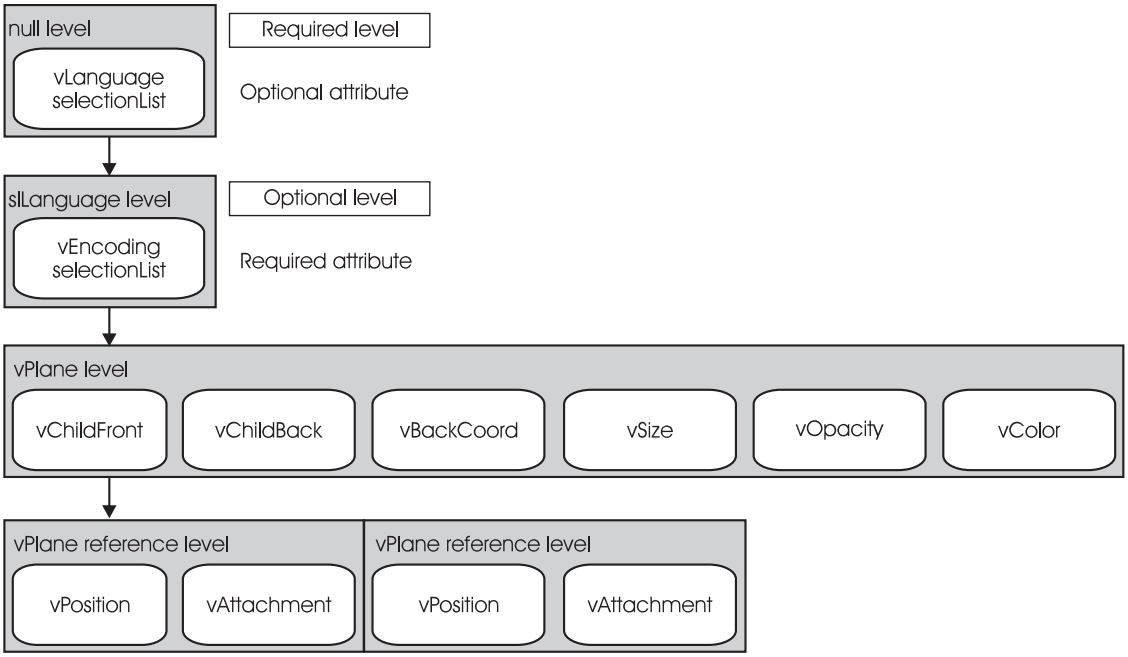
- Render that child
- Request content (attachment) information of next child
- and so forth until no more children exist

5. There are a number of ways to request content data for children, one or more at a time, whether the child list is known or unknown. The following methods are applicable to access the child surfaces in a child list:
   - Use ***vChild*** with one or more reference names to access the various child surfaces.
   - Use ***vFrontFirst*** or ***vBackFirst*** to access the first child surface in the list.
   - Use ***vFrontLast*** or ***vBackLast*** to access the last child surface in the list if appropriate.
   - Use ***vChildNext*** with ***vNumber*** equal to one or more to access the next one or more child surfaces in the list. This cannot be the first access to a child surface in this child list.
   - Use ***vChildNext*** with ***vNumber*** zero or undefined to access the remaining child surfaces in the list. This also cannot be the first access to a child surface in this child list.
6. Random access to surfaces in the child list is done by:
   - Using the surface name as the value to the ***vChildFront*** or ***vChildBack*** attribute in a content request. The surface name is returned in the content reply. In the case of vPlane, attachment information is also returned.
   - Using the child surface name to construct a description request. The surface description is returned in a description reply.
   - Using a content request to get the surface content of the child surface. The information is returned in a content reply.
   - Render the content information.
   - This process continues as required.
7. Serial access to surfaces in the child list is done by:
   - Using ***vFrontFirst*** or ***vBackFirst*** as the value to the ***vChildFront*** or ***vChildBack*** attribute in a content request on the parent surface. (The value ***vFrontLast*** or ***vBackLast*** may be used as the value to these attributes if *childCount* is positive and ***vArbitrary*** key is given.) The surface name of the specified child surface in the child list is returned in the content reply along with the number of remaining child surfaces in the child list. In the case of the ***vPlane*** encoding, attachment information is also returned.
   - Using the child surface name to construct a description request. The surface description is returned in a description reply. A new job is initiated by issuing a SurfaceRequestMsg.
   - Using a content request to get the surface content of the child surface. The information is returned in a content reply.
   - Render the child surface content information. In the case of ***vPlane*** encoding, use the attachment information.

- Using the child surface name as the value to the ***vChildNext*** attribute in a content request on the parent surface. The surface name of the next child surface in the child list is returned in the content reply along with the number of remaining child surfaces in the child list. In the case of the ***vPlane*** encoding, attachment information is also returned.
- This process continues no child surfaces remain.

8. The values of *childCount* and ***vArbitrary*** or ***vSequential*** key indicate the methods of access to the various surfaces in the child list:
   - A small positive *childCount* value and ***vArbitrary*** key indicates that the child surfaces may be accessed in any manner: serially forward, serially backward, or randomly. Random access is made possible using the surface names in the reference list. Additionally inline content for some child surface may be included.
   - A small positive *childCount* value and ***vSequential*** key indicates that the child surfaces may only be accessed serially forward. The surface names are included, but do not permit random access.
   - A large positive *childCount* value and ***vArbitrary*** key indicates that the child surfaces may be accessed in either serially forward or serially backward.
   - A large positive *childCount* value and ***vSequential*** key indicates that the child surfaces may only be accessed serially forward.
   - A negative *childCount* value (regardless of the setting of the ***vArbitrary*** or ***vSequential*** key) indicates that the child surfaces may only be "discovered" serially forward.
   - A *childCount* value of zero means that no child surfaces exist in the child list. In the case of ***vPlane***, if both child lists are empty, the ***vPlane*** surface is only a rudimentary surface with a few attributes such as ***vColor*** and ***vSize***. In the case of ***vAssociation***, if the child list is empty, the surface is useless.
9. Access to the child surfaces in the list may be direct through the reference name or indirect through other mechanisms. Each child surface in the child list has a reference name and an order position in the list. An order is present for all child surface lists, whether the sending appliance is constrained to offer the surfaces in that order (***vSequential***) or is capable of offering the surfaces in any order (***vArbitrary***). Each child surfaces in a ***vPlane*** surface also has a position on the surface (***vPosition***) and a method of attachment (***vAttachment***). Direct access of description or content information from child surfaces is always through the reference name. Indirect access of description or content information from child surfaces is though moving forward or backward serially through the list of children.

Hierarchy Diagram for vPlane



Note that the ***vLanguage*** attribute and the ***vLanguage*** level are both optional where as the null level and ***vEncoding*** attribute are both required. If the ***vLanguage*** attribute is omitted so is the ***vLanguage*** level; the ***vEncoding*** attribute then appears in the null level.

Keywords for Attributes and Values Defined for vPlane

Table A.31: Attributes

Attribute	Encoded format
vLanguage	
vEncoding	
vChildFront	
vChildBack	
vBackCoord	
vSize	
vOpacity	
vColor	
vChild	
vChildNext	

Table A.31: Attributes

vNumber	
vNames	
vRemaining	
vPosition	
vAttachment	

Table A.32: Values

Value	Encoded format
vEncoding	
vPlane	
vArbitrary	
vSequential	
vGray	
vSRGB	
vFirst	
vLast	
vFixed	
vFloating	

Examples for vPlane

vPlane Example #1

This example represents a one page plane, 8.5 inches by 11 inches, with two children (which are probably images. These child images of this plane are not included in this example). The child content information is included in line.

	vEncoding	vPlane
vPlane	vChildFront	2 vArbitrary 4 5
vPlane	vSize	(612000,792000)
vPlane.4	vPosition	(72000,360000)
vPlane.4	vAttachment	vFixed
vPlane.5	vPosition	(612000,612000)
vPlane.5	vAttachment	vFixed

vPlane Example #2

This example represents one page, two planes, one with two children, and the “back side” with one child. The child content information is *not* included in-line.

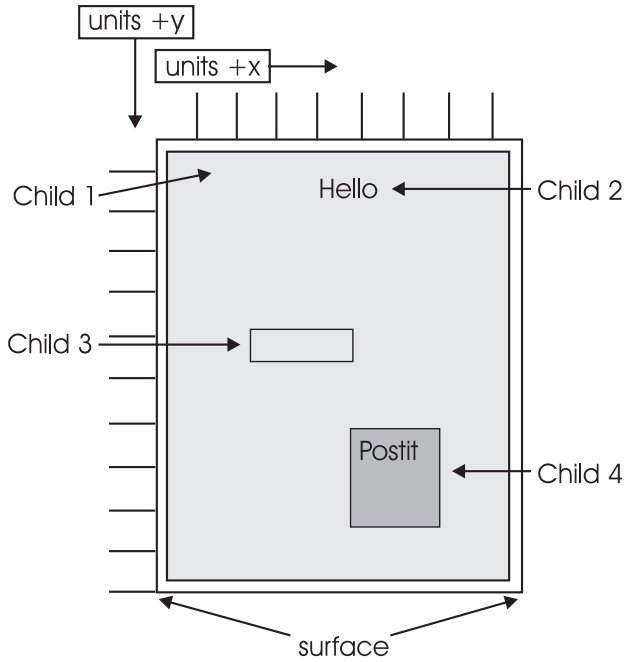
	vEncoding	vPlane
vPlane	vChildFront	2 vArbitrary 6 7
vPlane	vChildBack	1 vArbitrary 8
vPlane	vBackCoord	0
vPlane	vSize	(8500,11000)

vPlane Example #3

The following is an example of a plane representing a scanned page, 8.5 inches by 11 inches, with one annotation, a yellow highlight, and a red sticky type note with text. The latter three would have to have been added to the original scanned image, so the image is not being generated dynamically, and the number of children is known. Figure A.9 is a graphical representation of this image showing the children.

	vEncoding	vPlane
vPlane	vChildFront	4 vSequential 1 2 3 4
vPlane	vSize	(612000,792000)
vPlane.1	vPosition	(90000, 90000)
vPlane.1	vAttachment	vFixed
vPlane.2	vPosition	(288000, 72000)
vPlane.2	vAttachment	vFixed
vPlane.3	vPosition	(900000, 360000)
vPlane.3	vAttachment	vFixed
vPlane.4	vPosition	(360000, 540000)
vPlane.4	vAttachment	vFloating

Figure A. 9 Graphical representation of a plane with children



This is the first child of the parent page structure. It represents one image.

	vEncoding	vImage
vImage	vSize	(612000,792000)
vImage	...	
	Other Attributes l l l	*Other Values*

This is the second child of the parent page structure. It represents one of the text annotations.

	vEncoding	vText
vText	vSymbolSet	vAscii
vText	...	
	Other Attributes l l l	*Other Values*

This is the third child of the parent page structure. It represents the yellow highlighted area which is offered in two color models.

	vEncoding	vPlane
vPlane	vSize	(144000, 14400)
vPlane	vColor	255 255 0
vPlane	vOpacity	110

This is the forth child of the parent page structure. It represents a separate, red sticky type note.

	vEncoding	vPlane
vPlane	vChildFront	1 vArbitrary 5

vPlane	vSize	(144000, 144000)
vPlane	vColor	255 0 0
vPlane.5	vPosition	(0,7200)
vPlane.5	vAttachment	vFloating

Encoding for vAssociation

The ***vAssociation*** encoding is primarily designed to represent/describe the following types of information:

- Documents
- Any correlation between similar and dissimilar encoding types

Definition for vAssociation

The ***vAssociation*** encoding is a top level encoding, and as defined, contains children in order to represent complex information types. It is primarily used to describe the relationship between the children that are also top level encodings.

In a less abstract form, the association could be used to describe a relationship between multiple pages, a document. Moving back one level from the description of pages, an association could represent a folderthath applies a relationship between documents, which documents apply a relationship to the pages of the documents.

Surface Description for vAssociation

This table describes the layout of a ***vAssociation*** surface description.

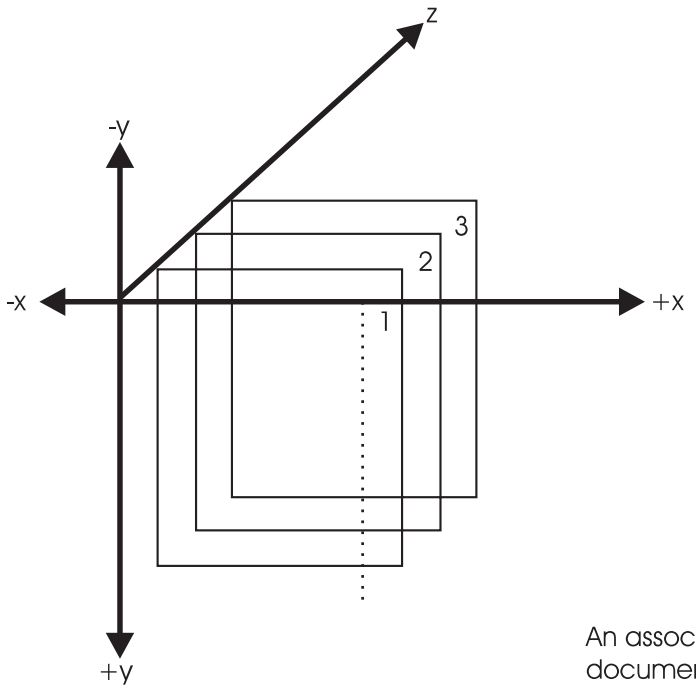
Table A.33: Description Block

Encoding Hierarchy	Attribute	Value	Comments
Decision Path: null	Optional: ***vLanguage***	*languageSelectionList* derived from: emListType	The ***vLanguage*** attribute may be included if reference to language and country is desired. A default language value is not specified.
Decision Path: [***vLanguage***]	Required: ***vEncoding***	*encodingSelectionList* derived from: emListType Definition: **vAssociation |** Additional: ***vImage*** | ***vText*** | ***vFile*** | ***vPlane*** | emStringType	A ***vAssociation*** provides an associate relationship between child surfaces. It has no attributes of its own. This section defines the ***vAssociation*** encoding specifically. The other encodings, ***vImage***, ***vText, vFile***, ***vPlane,*** and any unique emStringType encodings may also be present.

Table A.33: Description Block

Encoding Hierarchy	Attribute	Value	Comments
Decision Path**:** [***vLanguage.***] ***vAssociation***	Required: ***vChild***	*childCount* {***vArbitrary*** | ***vSequential***} [*reference* [*reference* [...]]] otherwise: *anyReferenceList* derived from: emListType	Provides a reference list which links child surfaces together. The *childCount* value indicates the number of known child surfaces. It may be negative, zero, or positive. Small positive values of *childCount* are paired with a matching number of references. Other values are paired with no references. If there are limitations on the order in which the sending appliance can send the surfaces referenced, the ***vSequential*** keyword is used, otherwise the keyword is ***vArbitrary***. This allows the receiving appliance to fully control the order of surfaces requested.

Figure A. 10 Graphical view of a vAssociation encoding representing a document



An association representing a document. The children (1, 2, 3) represent pages, and the "z" order of the pages determines the sequence in which the pages are "rendered".

Description Request Block for vAssociation

While it is very unlikely that the number of options available for representing a given ***vAssociation*** surface would ever exceed the available description size, description request and description reply messages are specified.

A description request is constructed by listing each attribute from the description already received along with the selection from each selection list. The encoding hierarchy is not used. Thus the receiver identifies the selections made by the receiver while processing the surface description.

The structure of a description request and the corresponding description reply are tightly coupled because the decision paths which form the encoding hierarchy of the description reply dependent on the attributes included in the description request.

Table A.34: Description Request Block

Encoding Hierarchy	Attribute	Value	Comments
(Not Used)	Optional: ***vLanguage***	***sl***Language	Include if available.
	Required: ***vEncoding***	***vAssociation***	Include if available.

Description Reply Block for vAssociation

A description reply is used to deliver additional description information that was not included in any previous surface description. A description reply is only sent in response to a description request.

The reply is formed by processing the request, recognizing the selections and therefore the partial decision path, and building a reply based on the hierarchy below the selected level. The decision path which forms the encoding hierarchy is dependent on the attributes included in the description request block. **Every level included in the description request is omitted from the description reply.** Otherwise the structure of the description reply block is identical to the structure of the original description block. The various selections from the selection lists are shown to be optional but are only so within the constraints of this request/reply pair.

For completeness, all of the attributes are listed here. It is extremely unlikely that ***vLanguage***, ***vEncoding***, or the other early attributes will ever be absent from the initial description (and thus forced into a description reply block).

Table A.35: Description Reply Block

Encoding Hierarchy	Attribute	Value	Comments
null	Optional: ***vLanguage***	*languageSelectionList*	As described earlier.
[***vLanguage***]	Required: ***vEncoding***	*keyStringSelectionList*	As described earlier.
[[***vLanguage.***] ***vAssociation***]	Required: ***vChild***	*childCount* {***vArbitrary*** | ***vSequential***} [*reference* [*reference* [...]]] otherwise: *anyReferenceList*	As described earlier.

Content Request for vAssociation

The user will want to use the content request for vAssociation when the child **name references** are not available. Those references can be requested through the Content Request using indirect child references (the values of ***vFirst*** and ***vLast***) to indicate that child information is being requested. The receiving device will provide the sending device with the child reference information. The sending device can then return to the receiving device, through the Content Reply, the information requested.

Table A.36: Content Request Block

Encoding Hierarchy	Attribute	Value	Comments
(Not Used)	Optional: ***vLanguage***	***sl***Language	Include if available.
	Required: ***vEncoding***	***vAssociation***	Selects the ***vAssociation*** encoding.
	Optional: ***vChildNext***	emStringType	Used when the name of the next child is unknown. It is used to request content for the next child in the sequence identified by the sending device. The ***vChildNext*** attribute can only be used if a previous child has been referenced. It may not be used if no previous child has been referenced. This attribute should be omitted if the ***vChild*** attribute is used with a reference name. The reference keyword is the child name prior to the child information being requested.

Table A.36: Content Request Block

Encoding Hierarchy	Attribute	Value	Comments
	Optional: ***vNumber***	emIntType	Used only when the ***vChild*** attribute is specified with a single reference, when the ***vChild*** attribute is specified with ***vFirst*** or ***vLast***, or when the ***vChildNext*** attribute is used. The attribute ***vNumber*** is associated with an *childCount* value indicating the number of references for which the receiving device is requesting content. When ***vNumber*** is used with a *reference* associated with the ***vChild*** attribute the items returned will include the referenced child. When ***vNumber*** is used with a *reference* associated with the ***vChildNext*** attribute the number of items returned will not include the referenced child but will include children following the reference child. If possible, the sender will send the content information for the children requested, in the order requested. If this attribute is missing, or if the value is zero, then the sender will send the content of as many children as possible from the reference made.

Table A.36: Content Request Block

Encoding Hierarchy	Attribute	Value	Comments
	Optional: ***vChild***	{*referenceList* | ***vFirst***|***vLast***} otherwise: *associationReferenceList* derived from: emListType	If a child has previously been referenced using ***vFirst*** *or* ***vLast***, these keyword values cannot be used again. See the ***vChildNext*** attribute. When a single reference is used, then it specifies a starting point for content information that is to be returned for one or more children. The number requested and the sequence, forward or backward from the reference is specified under the ***vNumber*** attribute below. If a reference list is specified, then it is an explicit request for the content of the children specified. The ***vNumber*** attribute is not used. The keyword ***vFirst*** is an indirect reference to the first child in the structure. This applies when the name references are specified as well as when the name references are unknown. The keyword ***vLast*** is used to refer to the last child in the same way. When, the number of children is unknown or access is sequential, the result of setting the child reference indirectly to the last one may not produce the results expected because the last one may not be accessible. The sending device retains no status of previous requests made, so once ***vFirst*** or ***vLast*** is used, subsequent requests should use an explicit reference returned in the previous content reply or the ***vChildNext*** attribute.

Content Reply for vAssociation

A content reply is used to deliver information concerning how child surfaces relate to the parent surface. In the case of ***vAssociation***, this includes the name of the child surfaces in the child list. The form of the content reply for both ***vPlane*** and ***vAssociation*** is similar to the description reply, in which encoding hierarchy, attributes, and values are returned. This is in contrast to the content reply for ***vImage***, ***vText***, and ***vFile***, in which raw content data is returned.

Table A.37: Content Reply Block

Encoding Hierarchy	Attribute	Value	Comments
Decision Path: null	Required: ***vNumber***	emIntType	Indicates the number of children in this content reply. If the value is greater than 0 it indicates the number of children in the reply. If the value is 0 there are no children in this reply.
Decision Path: null	Optional: ***vNames***	*referenceList* derived from: emListType	Gives the child references in the order requested. Optional only if the ***vNumber*** of children is 0. Each child reference must be unique. The # (pound sign) is a reserved symbol.
Decision Path: null	Required: ***vRemaining***	emIntType	Specifies the number of children for which a content request has not been made. A negative integer value may be specified indicating the number of children is unknown. A value of 0 indicates that there are no children remaining. A positive integer indicates the absolute number of children remaining.

Guidelines for an E-Material Encoding for vAssociation

1. Each reference associated with the ***vChild*** or ***vNames*** attributes must be unique. A mechanism has been provided in which a single surface may be referenced multiple times. If the receiving device wishes to implement this mechanism the receiving device would have the option of requesting the surface multiple times, or maintaining the reference data internally. A "#reference" (pound sign)(reference) may be included when the surface information is used multiple times. The "#reference" acts as an index to uniquely identify the child and child content. For example: If the surface name is blank, but the "blank" is to be rendered multiple times, the sender could provide the child reference names as blank#1, blank#2, blank#3. The receiving device then has the option of requesting the content information for the position and attachment as blank#1, blank#2 and blank#3 and requesting the surface "blank#1 or blank#2

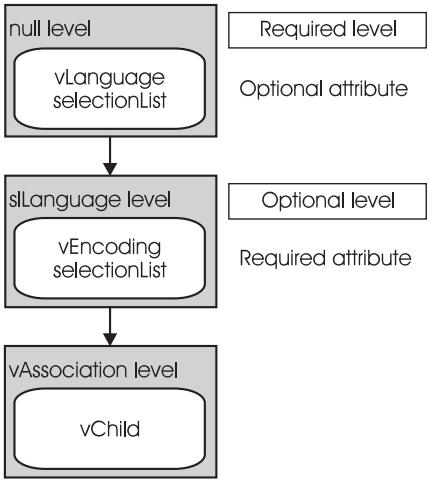
or blank#3" one time. The receiving device could also request the content information as blank#1, blank#2 and blank#3 and the surfaces as blank#1, blank#2 and blank#3, each as a unique surface.

2. The vAssociation encoding has been included to show an "association" between encoding types. The association can represent many different things, and those associations will grow as the number of encodings grows. Some devices may understand several levels of vAssociation as a child of a vAssociation encoding. The example where an association could represent a folder which applies a relationship between documents, which documents apply a relationship to the pages of the documents, may be understood by some devices and not understood by others. An attempt should be made by the devices to limit this type of nesting, but also to understand it when it is used. Devices that are not able to apply the nesting of non-renderable levels may choose to discard them and render the final renderable portion. Devices may also choose to interpret the encoding as one they are unable to decipher. Nested levels with non-renderable data requires memory, and the devices must determine the depth to which they are willing to work.
3. There are a number of ways to request content data for children, one or more at a time, whether the child list is known or unknown. The following methods are applicable to access the child surfaces in a child list:
   - Use ***vChild*** with one or more reference names to access the various child surfaces.
   - Use ***vFirst*** to access the first child surface in the list.
   - Use ***vLast*** to access the last child surface in the list if appropriate.
   - Use ***vChildNext*** with ***vNumber*** equal to one or more to access the next one or more child surfaces in the list. This cannot be the first access to a child surface in this child list.
   - Use ***vChildNext*** with ***vNumber*** zero or undefined to access the remaining child surfaces in the list. This also cannot be the first access to a child surface in this child list.
4. Random access to surfaces in the child list is done by:
   - Using the surface name as the value to the ***vChild*** attribute in a content request. The surface name is returned in the content reply.
   - Using the child surface name to construct a description request. The surface description is returned in a description reply. A new job is initiated by issuing a SurfaceRequestMsg.
   - Using a content request to get the surface content of the child surface. The information is returned in a content reply.
   - Render the content information.
   - This process continues as required.
5. Serial access to surfaces in the child list is done by:
   - Using ***vFirst*** as the value to the ***vChild*** attribute in a content request on the parent surface. (The value ***vLast*** may be used as the value to these attributes if *childCount* is positive and ***vArbitrary*** key is given.) The surface name of the specified child surface in the child list

is returned in the content reply along with the number of remaining child surfaces in the child list.

- Using the child surface name to construct a description request. The surface description is returned in a description reply. A new job is initiated by issuing a SurfaceRequestMsg.
- Using a content request to get the surface content of the child surface. The information is returned in a content reply.
- Render the child surface content information.
- Using the child surface name as the value to the ***vChildNext*** attribute in a content request on the parent surface. The surface name of the next child surface in the child list is returned in the content reply along with the number of remaining child surfaces in the child list.
- This process continues until no child surfaces remain.

Hierarchy Diagram for the vAssociation Level



Note that the ***vLanguage*** attribute and the ***vLanguage*** level are both optional where as the null level and ***vEncoding*** attribute are both required. If the ***vLanguage*** attribute is omitted so is the ***vLanguage*** level; the ***vEncoding*** attribute then appears in the null level.

Keywords for Attributes and Values Defined for vAssociation

Table A.38: Attributes

Attribute	Encoded format
vLanguage	
vEncoding	

Table A.38: Attributes

vChild	
vChildNext	
vNumber	
vRemaining	

Table A.39: Values

Value	**Encoded format**
vEncoding	
vArbitrary	
vSequential	
vFirst	
vLast	

vAssociation Examples

vAssociation Example #1

The following is an example of a structure representing a document consisting of three pages. The children can be impressed in any order the receiver chooses; however the logical ordering of the children from 1, 2, 3 should be fixed in that order.

	vEncoding	vAssociation
vAssociation	child	3 vArbitrary 1 2 3

General Guidelines for E-Material Encodings

The following is a list of rules for surface, attribute and value structure. Reference the E-Material encoding rules for each defined encoding for more specific guidelines.

1. Be a first-class citizen. E-material allows one to exercise a great deal of flexibility in the encodings. The flexibility lives within e-material to define encodings for a variety of devices. By defining an encoding of unusual or non-standard attributes one may achieve their goal of communication with "desired" devices, but may eliminate or defeat the purpose of a general communications mechanism to reach other devices using the same e-material class.
2. The process of decomposing an e-material encoding is not explicitly defined. The decomposition process must be based on the constraints of the device, it's memory, storage media, the amount of data manipulation it must perform, and so forth. Devices may want to process an e-material description in a depth first fashion. This is based on the fact that a depth first approach may limit the amount of memory used by the system. Associations usually need to be completed one page at a time.
3. Nested levels refers to a device building a hierarchy where top level encodings are placed as children of another top level encoding. An example of the type of nesting might be: ***vAssociation*** as a child of a ***vAssociation***, which is a child of a ***vAssociation***, which is a child of a ***vPlane***, which is a child of a ***vAssociation***, etc. In regards to nesting levels of encodings an attempt should be made by the devices to limit this type of nesting, but also to understand it when it is used. Devices that are not able to apply the nesting of non-renderable levels may choose to discard them and render the final renderable portion. Devices may also choose to interpret the encoding as one they are unable to decipher. Nested levels with non renderable data requires memory, and the devices must determine the depth to which they are willing to work.
4. Adding a new attribute to an encoding inherently creates a new encoding, and the encoding should be defined with a new keyword label.
5. Only one selection list should be presented at a encoding hierarchical level. Concise, simple and unambiguous encodings can ensure device communication.
6. If multiple selections are made available at within one hierarchical level, then it is assumed that all combinations are legal.
7. An attribute that is specified as optional, may include a value that is a selectionList from which the decision path is built. One of the values in the selectionList may be specified as the default value. The order in which the sending device builds the levels will determine the default. The default value must be the first value described in the selection list. These levels must be built correctly if the receiving device is to properly select the default.
8. There is an explicit order for attributes in the encoding hierarchy.
9. Adding a new selection value to an existing attribute in an encoding does not effect the definition of the encoding.
10. Attributes should not be multiply defined within an encoding, although the E-Material specification does not disallow their use. An appropriate exception may be the use of discrete

attribute-value pairs when a range is used within an encoding, even though the discrete attribute-value pair may be defined within the range. The use of a discrete attribute-value pair should have higher priority than the range and should precede the range in the description.

11. The order of the values in an attribute-value pair can be used to identify preference or priority of the sending device. It is at the discretion of the receiving device to determine which value to select, regardless of the sending devices preference.
12. The # (pound sign) is a reserved symbol when applied to reference names of the child surfaces. The # acts as a delimiter for specifying unique child names with indexes. This specification identifies the encodings in which references may use the # as a delimiter.
13. There is an explicit order for child surfaces. In a ***vAssociation***, one should expect to complete child surfaces one at time. The order is generally tied very strongly to the order of the physical media. In the ***vPlane***, the ***vPosition*** attribute plays a more significant role in indicating a physical ordering. In both cases the ordering often constrains access to child surfaces.
14. If the description is too large it must be stretched across description requests.
15. An entire hierarchical level must be contained within a single description or reply, and may not stretch across replies.
16. Default encodings must be supported.
17. Base encodings should be supported.
18. The sending device is not required to re-send description or content information if:

    - All of the description information related to the surface has been provided, regardless if it were in a single description surface or through the description surface and additional description replies.
    - All of the content information related to the surface has been provided, regardless if it were in-line or in multiple content replies.

    It is recommended that a sender maintain the description and content internally if possible and should do so until the surface has been released.
19. If a device can render content data in more than one format the surface description should identify those formats and a translation mechanism should be provided to convert from format to format.

Appendix B

E-Material Library API Reference

Identity: emPrepare

Description: Prepares a block of memory for use by other em functions.

Detail:
```
EMLIBAPI void emPrepare (Tbyte* em, Tu32 iSize);
```

Relevant files: appl_emlib.h

appl_emlib.c

Parameters and Return values: Parameter em is a pointer to a block of memory. It has to have at least iSize bytes available.

Parameter iSize is the size of the block. This is the number of bytes that the caller can use to write attribute/value pairs into the block. Once the size has been set for a given block, it cannot be changed.

Usage: This routine should be called before attributes can be added to the e-material block (using emAdd and emAddLevel). It clears/ the block of memory and adds a small e-material header to it.

If the function called on that block is already prepared (or has existing attributes in it), the block will be reset and all attributes already in it will be lost.

Requirements: The parameter em should point to a block of memory that has at least iSize bytes available.

See also: emAdd, emAddLevel

Samples:
```
Tbyte*  em;
em = Tmalloc(1024);
emPrepare(em, 1024);
```

Identity:	emSize
Description:	Returns the current *used* size of an e-material block.
Detail:	`EMLIBAPI Tu32 emSize (Tbyte* em);`
Relevant Files:	appl_emlib.h appl_emlib.c
Parameters Return Values:	Parameter em is a pointer to prepared em block. This function returns the number of bytes of data in the block. The maximum size is equal to the iSize parameter set when preparing the em block.
Usage:	Right after a call to emPrepare, emSize will return the number of bytes occupied by the em block header. When attributes are added, the size of the block will grow until it is full.
Requirements:	The parameter em should be a prepared em-block (emPrepare should have been called on it).
See also:	emPrepare, emAdd, emAddLevel
Samples:	

Identity: emMaxData

Description: Returns the maximum size of a data attribute that can be added to a given em block.

Detail:
```
EMLIBAPI Tu32 emMaxData (Tbyte* em, EMvalue vName);
```

Relevant files: appl_emlib.h

appl_emlib.c

Parameters
Return values: The parameter em is a pointer to prepared em block.

The parameter vName is the name of the data attribute that is going to be added. This parameter is necessary because different names occupy different amounts of memory in the em block, which impacts the amount left for the data value.

This function returns the number of bytes that can be added in a call to emAdd with vName attribute name, and a data value type.

Usage:
Requirements: This function is useful when the caller wants to fill up the remainder of a description block with binary data. This function will report how many bytes can be added in a data value, with the vName attribute name.

em should be a prepared em-block (emPrepare should have been called on it).

See also: emPrepare, emAdd, emData

Samples:
```
size = emMaxData(em, emString("my_data"));
emAdd(em, NULL, emString("my_data"), emData(size, pData);
```

Identity: emAdd

Description: Adds an attribute/value pair to an em block at a given level.

Detail:
```
EMLIBAPI Tbyte* emAdd (Tbyte* em, Tbyte* pLevel, EMvalue vName,
EMvalue vValue);
```

Relevant files: appl_emlib.h

appl_emlib.c

Parameters
Return values:

The parameter em is a pointer to prepared em block

The parameter pLevel is a pointer to level in the em block. The pointer can be NULL, in which case the attribute/value pair is added to the NULL level. Otherwise, the pointer can be set to the return value from a previous call to emAddLevel or emGetLevel.

The parameter vName is the name of the attribute. This can be a simple EMvalue parameter (any EMtype except emListType, emDataType, and emLevelType). It is generally easiest to call this function with a predefined value constant (vEncoding, vImage etc) or with one of the conversion functions (emKey, emString, emInt etc).

The parameter vValue is the value of the attribute. This can be any EMvalue parameter. emAdd will free any data associated with emListType values that allow the caller to supply a call to emList as a direct parameter to the emAdd function call. (It will free vValue.v.list.pValues.) This memory will be freed even if the function call fails. emAdd will copy data associated with emDataType values, and it will not free that data. (vValue.v.data.pData will not be freeõd.)

The function returns NULL if it failed to add the attribute. This will happen if there is not enough space available in the em block, or if the em block is not prepared with emPrepare or is malformed. If successful, the function returns a pointer into the em block where the attribute was added. The pointer is not useful for anthing except checking that the value was added.

Usage: Levels must be built in order. A level is closed when a new level is added. No more attribute-value pairs can be added to a closed level.

Requirements: The parameter should be a prepared em-block (emPrepare should have been called on it).

See also: emPrepare, emAddLevel

Samples:

Identity: emAddLevel

Description: Adds a sublevel to an em block at a given level.

Detail:
```
EMLIBAPI Tbyte*  emAddLevel (Tbyte* em, Tbyte* pLevel, EMvalue
vName);
```

Relevant files: appl_emlib.h

appl_emlib.c

Parameters
Return Values:

The parameter em is a pointer to prepared em block.

The parameter pLevel is a pointer to level in em block. The pointer can be NULL, in which case the attribute/value pair is added to the NULL level. Otherwise, the pointer can be set to the return value from a previous call to emAddLevel or emGetLevel.

The parameter vName is the name of the new sublevel. This can be a simple EMvalue parameter (any EMtype except emListType, emDataType, and emLevelType).

The function returns NULL if it failed to add the sublevel. If successful, the function returns a pointer into the em block that identifies the sublevel. This value can be used in subsequent calls to emAdd and emAddLevel (as well as emMatch, emGet, emGetLevel, etc.) to add attributes to the newly created level.

Usage: Levels must be built in order. A level is closed when a new level is added. No more attribute-value pairs may be added to a level that is closed.

Requirements: The parameter em should be a prepared em-block (emPrepare should have been called on it).

See also: emPrepare, emAdd

Samples:

Identity:	emKey
Description:	Converts an integer value into a keyword (emKeyType).
Detail:	`EMLIBAPI EMvalue emKey (Tu16 i);`
Relevant files:	appl_emlib.h appl_emlib.c
Parameters **Return Values:**	The parameter i is a keyword value. It should be one of the EM_xxx constants, for example, EM_ColorSpace.
Usage:	The parameter emKey returns an EMvalue with iType set to emKeyType and v.key set to the parameter specified. The function is typically used as a direct parameter to emAdd, emGet, or related functions. Instead of using the key function, a device can use the "v" constants. The keyword vColorSpace is equivalent to emKey(EM_ColorSpace), which obviously holds true for other keywords as well. Note: Device providers should not make up keyword values, but should only use the predefined values listed in the header file (EM_ColorSpace, for example). If a device provider feels like “inventing” new keywords, one can “invent” using strings less likely to collide with other devices’ private keywords.
Requirements:	none.
See also:	emAdd, emAddLevel, emGet, emGetLevel, emMatch.
Samples:	`emAdd(em, NULL, emKey(EM_Encoding), emKey(EM_Text));`

Identity: emString

Description: Converts an ASCII string into an EMvalue string (emStringType).

Detail:
```
EMLIBAPI EMvalue emString(char* s);
```

Relevant files: appl_emlib.h
appl_emlib.c

Parameters and Return values: The parameter i is a keyword value. It should be one of the EM_xxx constants, for example,. EM_ColorSpace.

Usage: The parameter emString returns an EMvalue with iType set to emStringType and v.string set to the parameter specified. The function is typically used as a direct parameter to emAdd, emGet, or related functions.

The string is not copied but the return value contains a pointer to it.

Requirements: none.

See also: emAdd, emAddLevel, emGet, emGetLevel, emMatch.

Samples:
```
emAdd(em, NULL, vData, emString("fest i gaden"));
```

Identity:	emInt emRange emIntPair emRangePair
Description:	Converts integer values into EMvalues.

Detail:	`EMLIBAPI EMvalue emInt (Ts32 i);`
	`EMLIBAPI EMvalue emRange (Ts32 iStart, Ts32 iEnd);`
	`EMLIBAPI EMvalue emIntPair (Ts32 i1, Ts32 i2);`
	`EMLIBAPI EMvalue emRangePair (Ts32 iStart1, Ts32 iEnd1, Ts32 iStart2, Ts32 iEnd2);`

Relevant files:	appl_emlib.h appl_emlib.c
Parameters and Return values:	The integer parameters are signed 32-bit integers.
Usage:	Each function returns an EMvalue with the iType field set to emIntType, emRangeType, emIntPairType, or emRangePairType.
Requirements:	none.
See also:	emAdd, emAddLevel, emGet, emGetLevel, emMatch.
Samples:	

Identity: emData

Description: Converts binary data into an EMvalue binary data (emDataType).

Detail:
```
EMLIBAPI EMvalue emData (Tu32 iSize, Tbyte* pData);
```

Relevant files: appl_emlib.h
appl_emlib.c

Parameters:

Return values:

Usage: The data is not copied, so the caller should keep the data around until the return value is used in a call to emAdd.

Requirements: none.

See also: emAdd, emAddLevel, emGet, emGetLevel, emMatch.

Samples:

Identity: emList

emListArray

Description: Converts a list of EMvalues to a list EMvalue (emListType).

Detail:

```
EMLIBAPI EMvalue emList (Tu32 iCount, ...);
EMLIBAPI EMvalue emListArray (Tu32 iCount, EMvalue* pValArray);
```

Relevant files: appl_emlib.h

appl_emlib.c

Parameters:
Return values: The parameter iCount is the number of parameters that follow this one, that is, the number of elements in the list.

Any other parameters are included in the iCount number of parameters, type EMvalue. All parameters should be simple values, that is, any type except emListType, emDataType, and emLevelType.

The parameter *pValArray is a pointer to a prebuilt array of values.

This function eturns a value of type emListType. The caller should free v.list.pValues of the return value when it is no longer needed.

Usage: The function has a variable number of arguments. The first parameter is the number of arguments to follow, and the following arguments should all be of the EMvalue type. This function is useful if the list consists of a well-known small number of elements. If the list is complicated and its size varies depending on some algorithm, emListArray is a more appropriate function to use.

The function emList allocates memory for the list and copies the values into that list. This memory should be freed by the caller. If passed directly to emAdd, that function will free the list.

The difference between emList and emListArray is that the former takes the list elements as variable number of arguments and the latter takes a prebuild array of values as argument. This function allocates new memory for the values and copies the values from pValArray.

Requirements: none.

See also: emAdd, emGet, emFreeValue

Samples: This example adds a vChildNames attribute to the em block. The value is a list of four elements: an integer (2), a keyword (*vSequential*) and two strings ("page1" and "page2").

```
emAdd(em, NULL, vChildNames,
emList(4, emInt(2), vSequential, emString("page1"),
emString("page2")));
```

Identity: emFreeValue

Description: Free the data associated with a value.

Detail:
```
EMLIBAPI void emFreeValue (EMvalue vValue);
```

Relevant files: appl_emlib.h
appl_emlib.c

Parameters:
Return values:

Usage: This is typically used in connection with a call to emGet. If the caller does not expect a list value when looking up the value of a given attribute, the given attribute still needs to free the data associated with the list that is returned. emFreeValue is an easy way to ensure that all data returned by emGet is freed up properly.

Requirements: none.

See also: emGet

Samples:
```
if (emGet(em, NULL, vDepth, &depth))
{
    /* do stuff */
    emFreeValue(depth);
}
```

Identity: emMatch

Description: Compares a value to the value of an attribute in an em block at a given level.

Detail:
```
EMLIBAPI Tbool   emMatch (Tbyte* em, Tbyte* pLevel, EMvalue
vName, EMvalue vValue);
```

Relevant files: appl_emlib.h

appl_emlib.c

Parameters:
Return values:

The parameter em is a pointer to a prepared em block.

The parameter pLevel is a pointer to level in em block. The pointer can be NULL, in which case the attribute is matched at the NULL level. Otherwise, the pointer can be set to the return value from a previous call to emAddLevel or emGetLevel.

The parameter vName is the name of the attribute in the em block that the caller is matching against. emMatch will look up value value of the attribute in the em block and compare vValue to it.

The parameter vValue is the value with which the caller is matching. If the value is complex (emListType, emDataType, or emLevelType), the function will always return false.

This function returns a 1 if vValue is successfully matching the value of vName; otherwise it returns a 0. See emCmp for a more detailed specification of when a compare results in true or false.

Usage: In a lot of cases this function can replace the need to call emGet. It is useful for checking if a value is part of the selection list of a particular attribute.

Requirements: none.

See also: emGet

emCmp

Samples: This call returns 1 if image is offered as an encoding (that is, if vImage is an element in a list value of the attribute vEncoding).

```
emMatch(em, NULL, vEncoding, vImage)
```

This function is equivalent to:

```
bReturn = 0;
if (emGet(em, pLevel, vName, &v))
{
    bReturn = emCmp(vValue, v);
    emFreeValue(v);
}
return bReturn;
```

Identity: emGet

Description: Gets the value of an attribute from an em block at a given level.

Detail:

```
EMLIBAPI Tbyte*  emGet (Tbyte* em, Tbyte* pLevel, EMvalue vName,
EMvalue* pValue);
```

Relevant files: appl_emlib.h

appl_emlib.c

Parameters:
Return values:

The parameter em is a pointer to a prepared em block.

The parameter pLevel is a pointer to level in the em block. The pointer can be NULL, in which case the attribute is gotten from NULL level. Otherwise, the pointer can be set to the return value from a previous call to emAddLevel or emGetLevel.

The parameter vName is the name of the attribute that the caller is extracting from the em block.

The parameter pValue is a pointer to a variable of type EMvalue. If the attribute vName is found at the given level, emGet will write the value to this location. If the attribute is not found, this location will be untouched. If the value returned is a list (emListType), the caller should free the data associated with that list (pValue->v.list.pValues). An easy way to do this is to call emFreeValue(*pValue).

If the attribute is found, a pointer to its location in the em block is returned. Otherwise, emGet returns NULL. The pointer is not useful for anything except checking that the value was found.

Usage: This function looks up an attribute and, if found, converts its value to an EMvalue and writes this to the location pointer to by pValue. If the value is a list (emListType), emGet allocates memory for its elements which should be freed by the calling function. If the value is data (emDataType), emGet returns a pointer into the em block that should *not* be freed by the caller.

Requirements: none.

See also: emGetLevel, emMatch

Samples:

Identity: emGetLevel

Description: Gets a level with the name vName from an em block at a given level.

Detail:
```
EMLIBAPI Tbyte*  emGetLevel (Tbyte* em, Tbyte* pLevel, EMvalue
vName);
```

Relevant files: appl_emlib.h

appl_emlib.c

Parameters:
Return values:

The parameter em is a pointer to a prepared em block.

The parameter pLevel is a pointer to level in the em block. The pointer can be NULL, in which case the level is gotten from the NULL level. Otherwise, the pointer can be set to the return value from a previous call to emAddLevel or emGetLevel.

The parameter vName is the name of the sublevel that the caller is getting.

This function returns a pointer to the sublevel with the name vName, if it is found in the em block at the given level. Otherwise, it returns NULL. The pointer can be used in calls to emGet, emMatch, emGetLevel, and (in some cases) emAdd and emAddLevel.

Usage: Gets a level with the name vName from an em block at a given level. If found, the function returns a pointer to the level that can be used in subsequent calls to emGet, emGetLevel, emGetNext etc.

Requirements: none.

See also: emGet, emMatch

Samples:

Identity: emCmp

Description: Compares vSimple against vComplex, and returns 1 if they match.

Detail:
```
EMLIBAPI Tbool emCmp (EMvalue vSimple, EMvalue vComplex);
```

Relevant files: appl_emlib.h
appl_emlib.c

Parameters: The parameter vSimple must be of type emKeyType, emStringType, emIntType, emIntPair, emRangeType, or emRangePair. The parameter vComplex can be any of those listed or emListType.

Return values: This function returns true (1) if vSimple is contained (in some sense) or equal to vComplex. Otherwise it returns 0.

Usage: Compares vSimple against vComplex, and returns 1 if they match. the parameter vComplex can be a list, in which case the function returns 1 if at least one of the elements in the list matches vSimple. The table below shows how the comparison is done:

vSimple	vComplex	Returns true (1), only if
(any)	`emListType`	.. vSimple matches at least one of the elements in vComplex.
`emKeyType`	`emKeyType`	.. the two keywords are identical.
`emStringType`	`emStringType`	.. the two strings are identical.
`emIntType`	`emIntType`	.. vSimple is equal to vComplex.
`emIntType`	`emRangeType`	.. vSimple is equal to one of the elements identified by the range.
`emIntPairType`	`emIntPairType`	.. the first elements in the pairs are equal *and* the second elements in the pairs are equal.
`emIntPairType`	`emRangePairType`	.. the first element in vSimple matches the first range in vComplex *and* the same is true for the second element and range.
`emRangeType`	`emRangeType`	.. the vSimple range is entirely included in the vComplex range.
`emRangePairType`	`emRangePairType`	.. both the first and the second vSimple range are entirely included in the first and second vComplex range, respectively.
(any other combination of types)		.. returns false (0).

Requirements: none.

See also: emGet, emMatch

Samples: Each of these expressions is true; evaluation using emCmp yields the value at the right.

```
(emCmp(emInt(1), emInt(2))                       == 0)
(emCmp(emInt(100), emRange(100,600))             == 1)
(emCmp(emIntPair(1,2), emRangePair(0,10,5,20)) == 0)
(emCmp(vEncoding, vEncoding)                     == 1)
(emCmp(vEncoding, emString("encoding"))          == 0)
(emCmp(emInt(42), emString("42"))                == 0)
```

Identity: emGetNext

Description: Gets the next attribute-value pair from level pLevel of the block em.

Detail: EMLIBAPI Tbyte* emGetNext (Tbyte* em, Tbyte* pLevel, Tbyte* pAttr, EMvalue* pName, EMvalue* pValue);

Relevant files: appl_emlib.h

appl_emlib.c

Parameters: The parameter em is a pointer to a prepared em block.

Return values: The parameter pLevel is a pointer to a level within the em block. A value of NULL indicates the NULL level, otherwise a return value from emGetLevel, emGetNext, or emAddLevel should be used.

The parameter pAttr is a pointer to the last attribute found. To search for the first attribute in a level, this parameter is set to NULL. Otherwise it is set to the return value of a previous call to emGetNext.

The parameter pName is the target for the attribute name. If an attribute was found, emGetNext will write the name of that attribute to the location specified by pName.

The parameter pValue is the target for the attribute value. If an attribute was found, emGetNext will write the value of that attribute to the location specified by pValue. If the type of the value is emListType, the caller should free the data associated with the list (for example, pValue->v.list.pValues). Calling emFreeValue(*pValue) is a convenient way of doing this.

If an attribute was found, the function returns a pointer into the em block identifying the attribute. This value can be used to find the next value by passing it in the pAttr parameter in a subsequent call to emGetNext. When there are no more attributes at a given level, emGetNext returns NULL.

Usage: Gets the next attribute-value pair from level pLevel of block em. If pAttr is NULL, the function will get the first attribute at the specified level. Otherwise it will return the attribute-value pair immediately following the attribute pointed to by pAttr. The function returns a pointer to the attribute found as well as the attribute name (*pName) and its value (*pValue). The return value can be used in a subsequent call to emGetNext() to go through all attributes of a level one by one. This is the only way a caller can get to attributes with unknown names.

Requirements: none.

See also: emGet, emGetLevel

Samples:

Identity: emPrint

Description: Prints out a nicely formatted version of an em block's contents.

Detail:
```
EMLIBAPI void emPrint (Tbyte* em);
```

Relevant files: appl_emlib.h
appl_emlib.c

Parameters:

Return values:

Usage: The function uses the LOG macro for printing, so depending on how that macro is defined the output might go to stdout, a file, or be lost forever.

Requirements: #define OPUS_DEBUG

See also: emPrintValue

Samples:

Identity: emPrintValue

Description: This function sprints a single value to the buffer pBuffer.

Detail:
```
EMLIBAPI void emPrintValue (EMvalue v, char* pBuffer);
```

Relevant files: appl_emlib.h
appl_emlib.c

Parameters:
Return Values:

Usage: This function sprints a single value to the buffer pBuffer. The buffer should have at least 80 bytes available for writing. If the value is binary data, only the first couple bytes of the data will be sprinted.

Requirements:

See also: emPrint

Samples:

Appendix C

Activity Manager Reference

Part of the Device Independent API

Identity:	amInitialize
Description:	This routine initializes the activity manager, prepares the event queue and sets other internal variables.
Detail:	`DEVAPI Tstatus amInitialize(Tu16 iSubAddress)`
Relevant Files:	am.c tops.h tops_am.h tops_device.h
Parameters:	The parameter Tu16 iSubAddress is the sub-address for the device. On a machine running more than one logical appliance, the sub-address must be different for each device.
Return Values:	This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h discussion in Chapter 1 for more information.
Usage:	This routine initializes the activity manager and lower layers. This routine should be called before any of the activities are initialized.
Requirements:	None.
See also:	amShutdown
Samples:	None.

Identity: amShutdown

Description: This routine shuts down the activity manager and frees any associated resources.

Detail:
```
DEVAPI Tstatus amShutdown(void)
```

Relevant Files: am.c
tops.h
tops_am.h
tops_device.h

Parameters:
Return Values: This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h in Chapter 1 for more information.

Usage: This routine shuts down the activity manager. This will end all active sessions, shut down all activities, and delete pending events.

Requirements: The function amShutdown can only be called after amInitialize returns successfully. Only amInitialize can be called after the AM is shutdown. A device will not receive any events after the AM is shutdown, however, a remote device may receive events generated by the shutdown (for example, amSessionEndEvent, psInGoneEvent, asSurfaceGoneEvent, etc.).

See Also: amInitialize

Samples: None.

Identity: amCreateBuf

Description: This function will create a buffer and leave enough header space for information that the infrastructure may add to the message.

Detail:

```
DEVAPI BD* amCreateBuf(Tu16 iSize)
```

Relevant Files: am.c

appl_buf.h

tops.h

tops_am.h

tops_device.h

Parameters:
Return Values: The parameter Tu16 iSize is the amount of useable space in the returned buffer. The actual buffer created will be larger to allow space for header information, but iSize is the amount of memory available to the device. iSize is limitied, depending on what the buffer will be used for. For e-material descriptions (all functions using buffers other than psSendContent), iSize is limited to the constant AM_MaxDescriptionSize. For buffers created to use in psSendContent calls, iSize is limited to the iMaxBufferSize parameter returned to the device in the previous psContentRequestEvent. Note that the above are limits; a buffer should generally be created with the smallest iSize possible.

On success, this function will return a pointer to a buffer. On failure, this function will return NULL.

Usage: This function will be used by the device to create buffers (BD*). These buffers send e-material descriptions and content via prRequestContent, psSendContent, prDescriptionEvent, etc.

Requirements: Buffer memory owned by the device should be freed by calling DestroyBuf (or amFreeEvent, if the buffer was passed to the device in an event). Buffers passed into the JetSend protocol stack (that is, in psSendContent) should not be freed by the device.

See Also: psSendContent

prContentEvent

psSendDescription

prDescriptionEvent

psStartJob

prJobStartEvent

prRequestContent

psContentRequstEvent

Samples:

```
BD *buildDesc1()
{
   EM newDesc;
   Tbyte *lvl;
   BD *newBuf;

   newBuf = amCreateBuf(AM_MaxDescriptionSize);
   emPrepare(BufDataAddr(newBuf), BufDataSpace(newBuf));
   newDesc = (EM)BufDataAddr(newBuf);

   emAdd(newDesc, NULL, vEncoding, vPlane);
   lvl = emAddLevel(newDesc, NULL, vPlane);
   emAdd(newDesc, lvl, vChildFront, emInt(1));
   emAdd(newDesc, lvl, vSize, emIntPair(8500,11000));
   emAdd(newDesc, lvl, vNames, emString(òzywiecó));

   return(newBuf);

}     /* END of function buildDesc1() */
```

Identity: amGetNextEvent

Description: This function polls the activity event queue and returns the next available event, if one exists.

Detail:
```
DEVAPI Tstatus amGetNextEvent(AMEVENT **pEvent)
```

Relevant Files: am.c

tops.h

tops_am.h

tops_device.h

Parameters:
Return Values: The parameter AMEVENT **pEvent returns a pointer to the event. If there are no events in the event queue, the function writes NULL to *pEvent. The device is responsible for freeing this memory (both the event structure and any memory that fields of the event structure point to).

If this function returns 0, it indicates that more is waiting to be done, and the function should be called again at some time. If this function returns TE_AM_LAST_EVENT, it signifies that the infrastructure has completed all current processing. On failure, the return value may be parsed to determine the location and cause of error. See tops.h in Chapter 1 for more information

Usage: This function spools the activity event queue. If there are any events available, a pointer to the first event will be returned in pEvent. The event is then deleted from the queue. If there are no events in the queue, the functions writes NULL to *pEvent. Calling amGetNextEvent will allow the lower layers to run.

Requirements: The AM must be initialized for this function to be called. The device is responsible for freeing both the event structure and memory pointed to by its members (strings, buffers, etc.). Calling amFreeEvent will free all associated memory.

See Also: amInitialize

amShutdown

amGetNextEvent

amFreeEvent

Samples:
```
AMEVENT *pEvent;
if( !amGetNextEvent(&pEvent)) {
    ...
    amFreeEvent(pEvent);
}
```

Identity: amFreeEvent

Description: This function frees all the event data associated with an event.

Detail:

```
DEVAPI Tstatus amFreeEvent(AMEVENT *pEvent)
```

Relevant Files: am.c

tops.h

tops_device.h

tops_am.h

Parameters: The parameter AMEVENT *pEvent is a pointer to the event to be freed.

Return Values: This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h in Chapter 1 for more information.

Usage: This function frees the event data associated with pEvent. This function will free the event data as well as any parameters that point to memory that should be freed. For example, calling amFreeEvent on a prContentEvent will free the event structure and the buffer containing the content data.

The function amFreeEvent will free all memory associated with an event. If a device wishes to keep memory passed up in an event (strings, buffers of e-material descriptions or content, etc.), there are a few choices.

1. The needed information can be copied, then amFreeEvent may be called.

2. The device can maintain pointers to all information in the event, then call Tfree to free only the event structure. The device can then call Tfree or DestroyBuf to free the saved memory when it is no longer needed.

3. The device can create a pointer to the information indicated by the pointer in the event structure. Then, the pointer in the event itself can be set to NULL, and amFreeEvent may be called to free the rest of the information in the event. The device can then call Tfree or DestroyBuf to free the saved memory when it is no longer needed.

A device can wait until the information is no longer needed before calling amFreeEvent.

Requirements: The parameter pEvent should point to an event returned via amGetNextEvent.

See Also: amGetNextEvent

Samples:

Identity: amStartSession

Description: This routine initiates a session with a specified device.

Detail:
```
DEVAPI Tstatus amStartSession (char *pAddress)
```

Relevant Files: am.c

tops.h

tops_device.h

tops_am.h

Parameters: char *pAddress is the network address of the target device. For example, ip:15.99.99.254 or fax:1-555-1212. The infrastructure copies the data pointed to by pAddress and does not free pAddress.

Return Values: This routine returns 0 on success.

Note: This does not mean a session has been opened successfully. It means the request made to the JetSend protocol stack to open a session has been made successfully.

If this call returns a 0, further status information will appear in the form of an amSessionStartEvent. On failure, the return value may be parsed to determine the location and cause of error. If the return value is non-zero, there will be no corresponding amSessionStartEvent. See tops.h in Chapter 1 for more information.

Usage: This routine initiates a session with the device at the specified address. The routine returns immediately, indicating the outcome of the request. If the routine returns 0, a corresponding amSessionStartEvent will occur, indicating whether or not the session was opened successfully. If the routine returns a non-zero value, there will be no corresponding amSessionStartEvent.

Requirements: The AM must be initialized in order for amStartSession to be called. To successfully establish a session with a remote device, the remote device must first have called amListenForSession.

See Also: amInitialize

amShutdown

amSessionStartEvent

amListenForSession

Samples:
```
amInitialize();
amStartSession();
/*Handle amSessionStartEvent*/
```

Identity: amSessionStartEvent

Description: This event notifies the device of the outcome of a session request. This event will occur on both the active and passive device when a session opens successfully.

Detail:
```
AMEVENT amSessionStartEvent(Tu32 iStatus, char *pAddress, char
*pAlias, SHANDLE hSesssion)
```

Relevant Files: am.c

tops.h

tops_device.h

tops_am.h

Event Structure Fields: The field Tu32 iStatus indicates the success or failure of the request. It can assume the following values.

AM_PASSIVE_OK - a session is established with a remote device, initiated by the remote device. For a passive side device, this is the only status value that can occur in an amSessionStartEvent.

AM_ACTIVE_OK - a session is established with a remote device, initiated by the local device.

AM_REJECTED - the session request was rejected by the remote device for some reason.

If iStatus is set to something other than one of the three above values, it indicates the session failed due to some problem on the local initiating device. The value can be parsed to determine the cause and location of the error. See tops.h in Chapter 1 for more information.

The field char *pAddress identifies the device with which a session has been opened. For the active device, this corresponds to the address in the amStartSession call.

The field char *pAlias is the alias address, if one exists. This pointer will be NULL on devices not supporting gateways.

The device is responsible for freeing any memory pAddress and pAlias point to, or for calling amFreeEvent to free this memory.

The field SHANDLE hSession is the session surface of the remote device. This is only valid if the iStatus value is AM_PASSIVE_OK or AM_ACTIVE_OK.

Usage: This event, with an iStatus of OK indicates a session has been established. Once this occurs, the various activities can begin using the session for communication. conRemoteComponentEvents, srDeviceStatusEvents, psContentRequestEvents, or prJobStartEvents may now occur, depending on which activities have been initialized. This communication may continue until the session is ended by the local device, via amEndSession, or by the remote device, indicated by an amSessionEndEvent. Note that regardless of which device starts the session, either device can end the session.

Requirements: On the active, or initiating side, an amSessionStartEvent will occur after a successful call to amStartSession. The amSessionStartEvent will indicate whether or not the session was opened successfully. On the passive side, an amSessionStartEvent will only occur when a session has been successfully opened.

See Also: amStartSession
amEndSession
amSessoinEndEvent

Samples:

```
amInitialize();
amStartSession();
/*Handle amSessionStartEvent*/
```

Identity: amEndSession

Description: This routine terminates an open session.

Detail:
```
DEVAPI Tstatus amEndSession(SHANDLE hSession)
```

Relevant Files: am.c
tops.h
tops_am.h
tops_device.h

Parameters: The parameter SHANDLE hSession specifies which session to end. This corresponds to a session handle returned in a previous amSessionStartEvent.

Return Values: This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. If the return value is non-zero, there will be no corresponding amSessionEndEvent. See tops.h in Chapter 1 for more information.

Usage: This routine terminates the session with the specific session surface. The routine returns immediately, and the result of the session termination is notified through an amEndSessionEvent, if this call returns 0.

Requirements: A device can only use this routine to end a session after the device has learned that the session has been started (via an amSessionStartEvent).

See Also: amSessionEndEvent
amStartSession
amSessionStartEvent

Samples:
```
amInitialize();
amStartSession();
     /* Handle amSessionStartEvent*/
...
amEndSession();
      /* Handle amSessionEndEvent*/
```

Identity: amSessionEndEvent

Description: This event notifies the device that a session has ended.

Detail:
```
AMEVENT amSessionEndEvent(Tu32 iStatus, SHANDLE hSession)
```

Relevant Files: am.c

tops.h

tops_am.h

tops_device.h

Event Structure Fields: The field Tu32 iStatus indicates how the session ended. It may take on one of two values.

AM_OK - the session ended because of a call to amEndSession

AM_TIMEOUT - the session ended because of a network timeout

The field SHANDLE hSession identifies which session has ended. This corresponds to the session handle returned in a prior amSessionStartEvent.

Usage: This event notifies the device that the session identified by the specified session surface has ended. The session handle is now invalid. After a successful call to amEndSession, both the active and passive device will receive this event.

Requirements: The AM must be initialized in order to receive this event. Additionally, it can only come after a corresponding amSessionStartEvent in which the status was OK.

See Also: amEndSession

amStartSession

amSessionStartEvent

Samples:
```
amInitialize();
amStartSession();
    /* Handle amSessionStartEvent*/
...
amEndSession();
    /* Handle amSessionEnd Event*/
```

Identity: amListenForSession

Description: This function allows the device to control whether or not new sessions can be established by remote devices.

Detail:
```
DEVAPI Tstatus amListenForSession(char *pNetwork, int iAccept)
```

Relevant Files: am_osp.c

am.c

appl.h

tops.h

tops_Devic.eh

tops_am.h

tops_osp.h

Parameters:
Return Values: The parameter char *pNetwork specifies the network type for which the function call is applied.

The infrastructure copies the string that pNetwork points to, so the device may free this memory.

If int iAccept is TRUE (non-zero), the JetSend protocol stack will accept new session requests on the specified network and issue the appropriate events to the device. If iAccept is FALSE (zero), the infrastructure will refuse all incoming session requests on the specified network.

Usage: A receiving (passive) device would call this to allow remote devices to establish sessions with it. After this function is called with an iAccept of 1, calling it again with an iAccept of 0 causes the local device to reject attempts by remote devices to establish sessions. A device can call this function with an iAccept of 0 after a number of sessions have been established to prevent new sessions from occuring, and can then call the function with an iAccept of 1 after one or more sessions have closed, allowing new sessions to again be established. In this way, passive devices can have control over the number of sessions they have open at any one time.

Requirements: This routine may be called anytime the activity manager is initialized. This routine must be called with a non-zero iAccept if the local device wishes to allow remote devices to establish sessions with the local device. (If amListenForSession is not called, the local device will not get an amSessionStartEvent unless it calls amStartSession itself, initiating an active session.)

See Also: amInitialize

amShutdown

amStartSession

amSessionStartEvent

Samples:

```
amInitialize();
amListenForSession(..., 1);
...
    /*Handle amSessionStartEvent*/
    /*Handle amSessionStartEvent*/
...
amListenForSession(..., 0);
/* Now, no new sessions can be established by remote
devices. Existing sessions are unaffected. */
       /*Handle amSessionEndEvent*/
amListenForSession(..., 1);
/* Now, new sessions can again be established by remote
devices. */
...
```

Identity:	amErrorEvent
Description:	This allows the network level of a device to pass an error buffer up past the JetSend protocol stack.
Detail:	`AMEVENT amErrorEvent(SHANDLE hSession, JOBID iJob, BD* errorBuf)`
Relevant Files:	am_oip.c tops.h tops_device.h tops_am.h tops_oip.h
Event Structure Fields:	The field SHANDLE hSession identifies the session on which the network error occured (if it exists). The field JOBID iJob identifies the job on which the error occured (if it exists). The field BD *errorBuf contains the original buffer passed up by the network layer. The device is responsible for freeing this memory.
Usage:	This event allows communication between the network layer (below the JetSend protocol stack) and the device (above the JetSend protocol stack). Information in the buffer remains untouched by JetSend protocol stack. The event will contain the session surface handle as well as the job identifier related to the error, if applicable.
Requirements:	This event can occur anytime the JetSend protocol stack is initialized.
See Also:	amInitialize amShutdown
Samples:	None.

Identity: amGetAliasAddress

Description: This routine gets the address by which a remote appliance has contacted a gateway.

Detail:
```
DEVAPI Tstatus amGetAliasAddress(SHANDLE hSession, char **pAlias)
```

Relevant Files: am_osp.c
appl.h
appl_tiapi.h
tops.h
tops_devic.eh
tops_am.h
tops_oiplib.h
tops_osp.h
tops_osppriv.h

Parameters:
Return Values: The parameter SHANDLE hSession is the session handle that identifies the remote device. This corresponds to a session handle returned in an amSessionStartEvent.

The parameter char **pAlias returns a pointer to the alias address string. This memory string should be freed by the device when no longer needed.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h in Chapter 1 for more information.

Usage: This function should not be used by anything other than a gateway appliance. This function does the same thing as amGetAddress, except that sessions within a gateway are created using an alias address (the phone number for a fax gateway, for example). This is the address returned from this function.

Requirements: A session must be established with a remote device, and the local device must be a gateway appliance.

See Also: amGetAddress
amSessionStartEvent

Samples: None.

Identity: amIsValidAddress

Description: This routine checks whether an address has a valid syntax.

Detail:
```
DEVAPI Tstatus amIsValidAddress(char *pAddress)
```

Relevant Files: am_osp.c
appl.h
appl_tiapi.h
tops.h
tops_devic.eh
tops_am.h
tops_oiplib.h
tops_osp.h
tops_osppriv.h

Parameters: The parameter char *pAddress is the string whose syntax will be checked.

Return Values: This routine returns 0 if the address has a valid syntax. A non-zero return value indicates the address is invalid. The non-zero return value can be parsed to see why the address is invalid. See tops.h in Chapter 1 for more information.

Usage: This routine checks whether an address has a valid syntax. See JMN Address Specification in Appendix L for more information.

Requirements: The AM must be initiailized for this routine to be called.

See Also: None.

Samples: None.

Identity: amEnumerateNetworks

Description: This routine will notify the device of the available networks that the transport layer supports.

Detail:
```
DEVAPI Tstatus amEnumerateNetworks(void)
```

Relevant Files: am_osp.c

tops_osp.h

Parameters:
Return Values: This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h in Chapter 1 for more information.

Usage: When the AM is initialized, amNetworkEvents will occur which inform the device of the available networks that the transport layer supports. If the device calls amEnumerateNetworks while the AM is initialized, it will cause these events to be passed up to the device again.

Requirements: The AM must be initialized for this function to be called. Calling this routine will cause an amNetworkEvent for each available network supported by the transport layer.

See Also: amNetworkEvent

amInitialize

amShutdown

Samples: None.

Identity:	amNetworkEvent
Description:	This event notifies the device of available networks supported by the transport layer.
Detail:	`AMEVENT amNetworkEvent(char *pNetworks, char *pAddress)`
Relevant Files:	am_osp.c tops.h tops_am.h tops_device.h tops_osp.h
Event Structure Fields:	The field char *pNetworks is a string representing the local network type: ip:, lb:, ux:, etc. The field char *pAddress a the local address for the given network type.
Usage:	This event can occur after the AM is initialized. One of these events will occur for each network supported by the available transport layer. Additionally, after a call to amEnumerateNetworks, one of these events will again occur for each supported available network.
Requirements:	This event may occur any time the AM is initialized. Zero or more amNetworkEvents will occur in response to an amEnumerateNetworks call. This event may also occur without having called amEnumeratedNetworks.
See Also:	amInitialize amShutdown amEnumerateNetworks
Samples:	None.

Identity: amAddGateway

Description: This routine is used by the device to inform TOPS of the presence of a gateway.

Detail:
```
DEVAPI Tstatus amAddGateway(char *pAddress, char *pNetworkType)
```

Relevant Files: am_osp.c

tops_device.h

tops_osp.h

Parameters:
Return Values:

The parameter char *pAddress is the OMNaddress of the gateway. It must have a sub-address, or the call to amAddGateway will fail. This infrastructure copies the data pointed to by pAddress and does not free pAddress.

The parameter char *pNetworkType is the type of gateway. At the moment, valid values are fax: and email:. The infrastructure copies the data in pNetworkType and does not free pNetworkType. New types may be added without any changes to TOPS, but other devices may need to know about the type if it is going to be usable.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h in Chapter 1 for more information.

Usage: A device uses this routine to add a gateway to the list of available networks. A successful call to amAddGateway will be followed by a corresponding amNewGatewayEvent

Requirements: The AM must be initialized. Following a successfull amAddGateway call, a corresponding amNewGatewayEvent will occur when the gateway is successfully added to the list of available networks.

See Also: amNewGatewayEvent

amRemoveGateway

amBroadcastGateway

Samples: None.

Identity:	amRemoveGateway
Description:	This function allows a device to remove a gateway from the list of available networks.
Detail:	`DEVAPI Tstatus amRemoveGateway(char *pReference)`
Relevant Files:	am_osp.c tops_device.h tops_osp.h
Parameters: **Return Values:**	The parameter char *pReference is the unique gateway reference returned in a previous amNewGatewayEvent. The memory pointed to by pReference is not freed by the infrastructure. This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h in Chapter 1 for more information.
Usage:	This function removes the requested gateway from the list of available networks. Each successfull call to amRemoveGateway will result in an amNetworkEvent with the network reference and a NULL address, which means that the gateway is no longer available.
Requirements:	The parameter pReference must correspond to the pReference parameter returned in a previous amNewGatewayEvent. The activity manager must be initialized (amInitialize).
See Also:	amAddGateway amNetworkEvent amNewGatewayEvent amBroadcastGateway
Samples:	None.

Identity: amBroadcastGateway

Description: This function sends a broadcast packet looking for gateways of a given type on a particular network.

Detail:
```
DEVAPI Tstatus amBroadcastGateway(char *pType, char
*pNetworkType)
```

Relevant Files: am_osp.c

tops_device.h

tops_osp.h

Parameters:
Return Values: The parameter char *pType is the type of gateway to look for (for example, fax: or email:). If NULL, all available gateways should respond.

The parameter char *pNetworkType specifies which network to send the broadcast on (for example, ip: or lb:). If NULL, all available network cards will be broadcast one time. Otherwise, the broadcast will be sent a maximum of four times for 90 seconds, or until the first response is received on the network.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h in Chapter 1 for more information.

Usage: This function will send a broadcast packet out looking for gateways. Any new gateways that respond and are not in the current list of available networks will cause an amNetGatewayEvent.

Requirements: The activity manager must be initialized (amInitialize).

See Also: amAddGateway

amRemoveGateway

amNetworkEvent

amNewGatewayEvent

Samples: None.

Identity: amNewGatewayEvent

Description: This event notifies the device that a request to add a gateway to the list of available networks has succeeded.

Detail:
```
AMEVENT amNewGatewayEvent(char *pAddress, char *pReference, Tu16
domain, Tu8 local, Tu32 status)
```

Relevant Files: am_osp.c

tops_device.h

tops_osp.h

appl_tiapi.h

Event Structure Fields: The field char *pAddress is the OMNaddress of the gateway. This corresponds to the pAddress in the corresponding amAddGateway call. The device should free the memory that pAddress points to.

The field char *pReference is the string that specifies how to reference this gateway. This value will be used in calls to amRemoveGateway for this gateway.

The field Tu16 domain indicates the domain type. This value corresponds to a list of Tdomain enumerated types in appl_tiapi.h. Currently, possible values are:

ETdomainINET ip:Internet domain:TCP, RMTP/UDP

ETdomainSIPX ipx:Novell domain:SPX, RMTP/IPX

ETdomainIrDA ir:Infrared domain:TTP

ETdomainUnix up:Unix domain: Unix paths

ETdomainLoopBack lb:For loopBack testingÑno transport

The field Tu8 local specifies whether the address is local or remote. Local is TRUE if it is a local address, FALSE if the address is a remote address.

The field Tu32 status is 0 on success, indicating the gateway specified in the amAddGateway call has been successfully added to the list of networks. On failure, the return value may be parsed to determine the location and cause of error. See tops.h in Chapter 1 for more information.

Usage: This event informs the device of the outcome of an amAddGateway request.

Requirements: This event can only occur while the AM is initialized. Additionally, this event will only occur after a successfull call to amAddGateway.

See Also: amInitialize

amShutdown

amAddGateway

amRemoveGateway

amBroadcastGateway

Samples: None.

Identity: amChangeDefaultSubAddr

Description: This function will update the default subaddress. This gives devices the ability to assign a default JIP when a subaddress is not specified.

Detail:
```
DEVAPI Tstatus amChangeDefaultSubAddr(Ts16 newSubAddr)
```

Relevant Files: am_osp.c

tops_device.h

tops_osp.h

appl.h

Parameters: The parameterTs16 newSubAddr is the subaddress that will become the global default settting.

Return Values: This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h in Chapter 1 for more information.

Usage: This function will update the default subaddress. If a device requests a session opened with ip:15.99.99.254 (note there is no subaddress), the subaddress connecting on the passive side will be the passive side's default subaddress. This gives devices (particularly multi-function devices) the ability to assign a default JIP when a subaddress is not specified.

Requirements: This cannot and should not be called before both ospSetup and amInitialize are called. ospSetup sets the global default subaddress DEVAULT_SUBADDR as defined in appl.h; amChangeDefaultSubAddr changes this initial setting.

See Also: None.

Samples: None.

Appendix D

Connection Activity Reference

Part of Device Independent API

Identity: conInitialize

Description: This routine initializes the contact activity, which provides the tools for PIN-based security. When devices start a session, the contact activity on each end will exchange self surfac**es**, structures with general information about a device; these self surfaces are similar to business cards that people exchange when they meet.

Detail:

```
DEVAPI Tstatus conInitialize(Tu16 iFetchMask)
```

Relevant Files: am_con.c

tops.h

top_device.h

tops_am.h

appl_em.h

appl_emlib.h

tops_oip.h

Parameters:
Return Values:

The parameter Tu16 iFetchMask specifies the components that requested automatically by the contact activity when new sessions are started. The mask is constructed by ORing component IDs. Legal component IDs are: SELF_LABEL, SELF_ICON, SELF_ENCODINGS, SELF_PIN.

Specifying SELF_PIN in the iFetchMask will result in each remote device receiving a conPINRequestEvent every time a session is established with the local device. The remote device will respond with conSendPIN, to either reject of fulfull the request. The local device will then receive the PIN or notification of the rejection via a conRemoteComponentEvent.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage:

Any device wishing to use the contact activity will call this routine. It is often called immediately following amInitialize, and must return successfully before any other contact activity routines can be called.

If a device requires the label and encodings but not the icon of other devices, it initializes the contact activity with conInitialize(SELF_LABEL | SELF_ENCODINGS). If a device is not interested in receiving any of the components of a remote device, it should call conInitialize(0).

Requirements:

This routine can only be called after a successful amInitialize and before an amShutdown. After conInitialize returns successfully, it should not be called again unless there has been a conShutdown.

See Also:

amInitialize

amShutdown

conShutdown

conPINRequestEvent

conSendPIN

conRemoteComponentEvent

Samples:

Identity: conShutdown

Description: This routines shuts down the contact activity and frees any associated resources.

Detail:
```
DEVAPI Tstatus conShutdown(void)
```

Relevant Files: am_con.c
tops.h
top_device.h
tops_am.h
appl_em.h
appl_emlib.h
tops_oip.h

Parameters and Return Values: This routine takes no arguments.
This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: This routine will be called when a device wants to shut down the contact activity and free its resources.

Requirements: This routine can only be called after a successful amInitialize and before an amShutdown. conInitialize must also return successfully before this routine can be called.

See Also: amInitialize
amShutdown
conInitialize

Samples:
```
conShutdown();
```

Identity: conSetComponent

Description: This routine sets a component of the local **self-surface**. The components of a self-surface are exchanged when two devices with initialize contact activities start a session. These components are used to exchange general information about each device, including the SELF_LABEL, SELF_ICON, and SELF_ENCODINGS.

Detail:
```
DEVAPI Tstatus conSetComponent(Tu16 iComponentId, void
*pComponent)
```

Relevant Files: am_con.c

tops.h

top_device.h

tops_am.h

appl_em.h

appl_emlib.h

tops_oip.h

Parameters:
Return Values: The parameter Tu16 iComponentId specifies which component is being set. Valid values are SELF_LABEL SELF_ICON, or SELF_ENCODINGS.

The parameter void *pComponent is the component itself. For SELF_LABEL, it is a (char*), which is a textual description of the device. For SELF_ICON, it is of type (EM), which is a pictorial representation of the device. The icon data is an e-material description with an in-line image. For SELF_ENCODINGS, it is of type (EM), which is a list of this device's preferred encodings whose format is specified in the E-Material Library Reference.

The JetSend protocol stack will free the component data when it is no longer needed. This happens when the component is changed or deleted, or the contact module is shut down.

Note: This means that for the SELF_LABEL, the (char*) cannot be a staticly allocated string. It should point to memory created via Tmalloc.

This routine returns 0 on success. On failure, the return value can be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: This routine will result in a conRemoteComponentEvent on the remote device if a session has been started, both sides have initialized contact activities, and the remote device is interested in this component.

Calling this routine for a component that has already been set initiates an update that results in a conRemoteComponentEvent on the remote device if the above three conditions are satisfied. Calling this routine with a NULL pComponent will delete the specified component and also result in a conRemoteComponentEvent on the remote device if the above three conditions are met. Calling this routine to set the SELF_PIN is invalid. Only conSendPIN should be used for the PIN, since this value will only be sent to the requesting device, and not every open session.

Requirements: This routine may be called any time after a successful conInitialize and before a conShutdown.

See Also:

conInitialize

conShutdown

conGetComponent

conRemoteComponentEvent

conRequestRemoteComponent

conReleaseRemoteComponent

conSendPIN

Samples:

Here is an example where the SELF_ICON component of a self-surface is set. This snippet could be used after amInitialize and conInitialize return successfully. See the E-material Primer and E-material Library Reference for an explanation of the following E-material functions.

```
EM getSelfIcon()
{
   EM encodings;
   Tbyte *lvl;

   encodings = (EM)Tmalloc(AM_MaxDescriptionSize);
   emPrepare(encodings, AM_MaxDescriptionSize);

   emAdd(encodings, NULL, vEncoding, vImage);
   lvl = emAddLevel(encodings, NULL, vImage);
   emAdd(encodings, lvl, vSize, emIntPair(200,200));
   emAdd(encodings, lvl, vColorSpace, vRGB);
   lvl = emAddLevel(encodings, lvl, vRGB);
   emAdd(encodings, lvl, vPixelDepth, emInt(24));
   lvl = emAddLevel(encodings, lvl, emInt(24));
   emAdd(encodings, lvl, vCompression, vNone);
   emAdd(encodings, lvl, vResolution, emIntPair(150,150));

   return(encodings);


}     /* END of function getSelfIcon() */

EM selfIcon = getSelfIcon;
conSetComponent(SELF_ICON, (void*) selfIcon);
```

Identity: conGetComponent

Description: This routine can be used to access the component of the local self-surfaces in case the device doesnõt keep track of them itself.

Detail:
```
DEVAPI Tstatus conGetComponent(Tu16 iComponentId, void
**pComponent)
```

Relevant Files: am_con.c

tops.h

top_device.h

tops_am.h

appl_em.h

appl_emlib.h

tops_oip.h

Parameters: The parameter Tu16 iComponentId specifies the component to be retrieved. Valid values are SELF_LABEL, SELF_ICON, or SELF_ENCODINGS.

Return values: The parameter void **pComponent is set to a pointer to the component itself. For SELF_LABEL, it is a (char*) which is a textual description of the device. For SELF_ICON, it is of type (EM) which is a pictorial representation of the device. The icon data is an e-material description with an in-line image. For SELF_ENCODINGS, it is of type (EM) which is a list of this deviceõs preferred encodings whose format is specified in the E-material Reference.

The JetSend protocol stack retains ownership of this memory, which is valid until the device calls conSetComponent again for that component or the contact module is shut down (at which point the memory is freed automatically by the contact activity).

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: See requirements.

Requirements: This routine may be called anytime after a successful conInitialize and before a conShutdown.

See Also: conInitialize

conShutdown

conSetComponent

conRemoteComponentEvent

conRequestRemoteComponent

conReleaseRemoteComponent

Samples:
```
EM selfEncodings;


conGetComponent(SELF_ENCODINGS, (void**)&selfEncodings);
```

Identity: conRequestRemoteComponent

Description: This procedure is used to request a component from the *self surface* of a remote device or a PIN from a remote device. A component include a label, icon or set of encodings.

Detail:
```
DEVAPI Tstatus conRequestRemoteComponent(SHANDLE hSession, Tu16
iComponentId)
```

Relevant Files: am_con.c

tops.h

top_device.h

tops_am.h

appl_em.h

appl_emlib.h

tops_oip.h

Parameters:
Return Values: The parameter SHANDLE hSession is the session handle for the remote device. This is a value returned in the amSessionStartEvent and the conRemoteComponentEvent.

The parameter Tu16 iComponentId specifies which component is to be requested. Valid values are SELF_LABEL SELF_ICON, SELF_ENCODINGS or SELF_PIN..

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: This routine will normally be used to request components that are not automatically fetched (as specified by the iFetchMask in conInitialize). The result of this routine will be returned through a conRemoteComponentEvent. Changes and deletions to the component will be notified as well, until the two devices end the session, either contact module is shut down, or the device calls conReleaseRemoteComponent. If conRequestRemoteComponent is called for a remote session that does not have a contact module initialized, the call will fail and return an error. This function may also be used by a device which wishes to fetch PINs only from specific sessions (and thus does not want to specify SELF_PIN in the conInitialize fetchmask), or to re-request PINs after receiving an invalid PIN or a rejection. Each call to conRequestRemoteComponent for the SELF_PIN component will result in a conPINRequestEvent on the remote device.

Requirements: A session must be established with the remote device. This will be marked by an amSessionStartEvent. Each side must also have its contact module initialized.

See Also: amStartSession

amSessionStartEvent

conInitialize

conShutdown

conSetComponent

conRemoteComponentEvent

conReleaseRemoteComponent

conPINRequestEvent

Samples:

For example, if the contact module is initialized to only request the SELF_LABEL and SELF_ICON, we may choose to ask for the SELF_ENCODINGS as well, after a session has started. Below, sessionOne is set to the session handle parameter which is returned in the amSessionStartEvent and in all conRemoteComponentEvents.

```
SHANDLE sessionOne;

sessionOne = event->data.conRemoteComponentEvent.hSession;
conRequestRemoteComponent(sessionOne, SELF_ENCODINGS);
```

Identity: conReleaseRemoteComponent

Description: This function is used to inform a remote device that we are no longer interested in receiving information about a specific component of the remote device's *self surface*. These components may include a label, icon or encodings.

Detail:
```
DEVAPI Tstatus conReleaseRemoteComponent(SHANDLE hSession, Tu16
iComponentId)
```

Relevant Files: am_con.c

tops.h

top_device.h

tops_am.h

appl_em.h

appl_emlib.h

tops_oip.h

Parameters :
Return Values: The parameter SHANDLE hSession is the session handle for the remote device. This is a value returned in the amSessionStartEvent and the conRemoteComponentEvent.

The parameter Tu16 iComponentId specifies which component is to be requested. Valid values are SELF_LABEL SELF_ICON, or SELF_ENCODINGS.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: This routine is used to inform a remote device that the local device is no longer interested in a given component. It would normally be called after a conRemoteComponentEvent for a component the local device has no further interest in. It may be called for a component originally specified in the conInitialize fetchmask or in a conRequestRemoteComponent call.

Requirements: A session must be established with the remote device. This will be marked by an amSessionStartEvent. Each side must also have its contact module initialized. Additionally, the component to be released must be a component which we have received via a conRemoteComponentEvent.

See Also: amStartSession

amSessionStartEvent

conInitialize

conShutdown

conSetComponent

conRemoteComponentEvent

conReleaseRemoteComponent

Samples:

If a conRemoteComponentEvent arrives that specifies the remote device's ICON, we may not be interested in this specific component for this remote device. We can call conReleaseRemoteComponent so we no longer receives updates for changes or deletions of this component.

```
SHANDLE sessionOne;

sessionOne = event->data.conRemoteComponentEvent.hSession;
conReleasetRemoteComponent(sessionOne, SELF_ICON);
```

Identity: conRemoteComponentEvent

Description: This event occurs when a component of the remote device's self surface arrives. These components are either automatically requested based on the iFetchMask in conInitialize or requested explicitly with conRequestRemoteComponent. Possible components include the self label, icon, encodings and pin.

Detail:
```
AMEVENT conRemoteComponentEvent(SHANDLE hSession, Tu16
iComponentId, void *pComponent)
```

Relevant Files: am_con.c

tops.h

top_device.h

tops_am.h

appl_em.h

appl_emlib.h

tops_oip.h

Event Structure Fields: The field SHANDLE hSession is the session handle used to identify the remote device.

The field Tu16 iComponentId specifies which component is being received. Valid values are SELF_LABEL SELF_ICON, SELF_ENCODINGS, or SELF_PIN.

The field void *pComponent is the component itself. For SELF_LABEL, it is a (char*) which is a textual description of the device. For SELF_ICON and SELF_ENCODINGS, it is of type (EM) which is a buffer descriptor. This buffer contains an e-material description of the icon or encodings, respectively. For SELF_PIN, it is a (char*) which is a character string sent by the remote device.

A NULL value for SELF_LABEL, SELF_ICON or SELF_ENCODINGS means the component has been deleted by its owner. A NULL value for SELF_PIN indicates the remote device rejected the PIN request.

It is the responsibility of the device to free the component data when it is no longer needed. This may be done explicitly, or via an amFreeEvent. When amFreeEvent is called, all memory held by the event is freed.

Usage: If a session is established with a remote device, and both deviceõs have an initialized contact module, the remote device will attempt to send any requested components (from either the conInitialize iFetchMask or conRequestRemoteComponent calls) it has available. When a component arrives, it causes an event to be posted on the AM event queue. A device calling amGetNextEvent will retrieve this event and have access to its data.

Requirements: A session must be established and each device must have its contact module initialized. Additionally, the remote device must have the requested componentes available.

See Also:

amStartSession

amSessionStartEvent

amGetNextEvent

amFreeEvent

conInitialize

conShutdown

conSetComponent

conRequestRemoteComponent

conReleaseRemoteComponent

Samples:

```
AMEVENT *tempEvent=NULL;
SHANDLE hSession;
Tu16 iComponentId;
char *remoteLabel=NULL;
BD *remoteEM=NULL;           /* This will hold ICON or ENCODINGS
*/


amGetNextEvent(&tempEvent);
if (tempEvent->iEvent == conRemoteComponentEvent)
{
    hSession = tempEvent->data.conRemoteComponentEvent.hSession;
    iComponentId = tempEvent-
>data.conRemoteComponentEvent.iComponentId;
    if (iComponentId == SELF_LABEL)
        remoteLabel = (char*) (tempEvent-
>data.conRemoteComponentEvent.pComponent);
    else
        remoteEM = (BD*) (tempEvent-
>data.conRemoteComponentEvent.pComponent);
}
```

Note: At this point, if we call amFreeEvent(tempEvent), either remoteLabel or remoteEM will become an invalid pointer, because the memory it points to is freed.

Identity: conPINRequestEvent

Description: As part of PIN based security, a device can request a PIN from another device, and then decide to end the session, or accept jobs, etc, based on the response. This event notifies the local device that a remote device has requested a PIN.

Detail: AMEVENT conPINRequestEvent(SHANDLE hSession, REQID iRequestId)

Relevant Files:
am_con.c
tops.h
top_device.h
tops_am.h
appl_em.h
appl_emlib.h
tops_oip.h

Event Structure Fields: The field SHANDLE hSession is the session handle used to identify the remote device.
The field REQID iRequestid is a unique identifier generated to keep track of this request. This id should be used when fullfilling the request via conSendPIN.

Usage: A remote device can request a PIN from any other device by specifying SELF_PIN in its conInitialize fetchmask or by calling conRequestRemoteComponent to request the PIN from a specific session. When a session is established, and the remote device requests a PIN, a conPINRequestEvent will occur on the local device. Before starting jobs, etc, the local device must respond to the conPINRequestEvent via conSendPIN. Calling conSendPIN will result in a conRemoteComponentEvent on the remote device.

Requirements: A session must be established with a remote device, and each side must have its contact module intialized. Additionally, the remote device must request the PIN component, via its conInitialize fetchmask or by calling conRequestRemoteComponent.

See Also:
conInitialize
conSendPIN
conRemoteComponentEvent

Samples: None.

Identity: conSendPIN

Description: This routine is used to respond to a PIN request.

Detail: DEVAPI Tstatus conSendPIN(REQID iRequestId, char *pin)

Relevant Files:
am_con.c
tops.h
top_device.h
tops_am.h
appl_em.h
appl_emlib.h
tops_oip.h

Parameters :
Return Values:
The parameter REQID iRequestId is used to identifiy which request is being satisified. This corresponds to the iRequestId returned in the conPINRequestEvent.

The parameter char *pin is the PIN of the remote device (identified by the hSession value in the corresponding conPINRequestEvent). If the pin is NULL, the remote device will treat this as a rejection of its PIN request.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: A remote device can request a PIN from any other device by specifying SELF_PIN in its conInitialize fetchmask or by calling conRequestRemoteComponent to request the PIN from a specific session. When a session is established, and the remote device requests a PIN, a conPINRequestEvent will occur on the local device. Before starting jobs, etc, the local device must respond to the conPINRequestEvent via conSendPIN. Calling conSendPIN will result in a conRemoteComponentEvent on the remote device.

If the local device doesnõt have the correct PIN, or wants to reject the PIN request, it should call conSendPIN with a NULL pin.

Requirements: A session must be established with a remote device, and each side must have its contact module intialized. Additionally, the remote device must request the PIN component, via its conInitialize fetchmask or by calling conRequestRemoteComponent. This routine should only be called in response to a conPINRequestEvent.

See Also:
conInitialize
conRequestRemoteComponent
conPINRequestEvent
conRemoteComponentEvent

Samples: None.

Appendix E

Push Send/Push Receiver Activity Reference

Part of the Device Independent API

Identity: psInitialize

Description: This routine initializes the sender side of the push activity, which is used by sending devices to send E-material.

Detail:
```
DEVAPI Tstatus psInitialize(void)
```

Relevant files: am_ps.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
tops_am_job.h
tops_oip.h
tops_oiplib.h

Parameters : This routine takes no arguments.

Return values: This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: Any device wishing to implement the send side of the push activity must call this routine. It will often be called immediately following amInitialize. It must return successfully before any other push send activity routines may be called.

Requirements: This routine may only be called after a successful amInitialize and before an amShutdown. After psInitialize returns successfully, it should not be called again unless there has been a successful psShutdown.

See also: amInitialize
amShutdown
psShutdown

Samples:

```
amInitialize(0);
psInitialize();
```

Identity: psShutdown

Description: This routine shuts down the push send activity and frees any associated resources.

Detail:
```
DEVAPI Tstatus psShutdown(void)
```

Relevant files: am_ps.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
tops_am_job.h
tops_oip.h
tops_oiplib.h

Parameters: This routine takes no arguments.

Return values:

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: This routine will be called when a device wants to shut down the push send activity and free its resources.

Requirements: This routine may only be called after a successful amInitialize and before an amShutdown. psInitialize must also return successfully before this routine may be called.

See also: amInitialize
amShutdown
psInitialize

Samples:
```
psShutdown();
```

Identity: psInArrivedEvent

Description: This event notifies the device that the *in surface* of a remote device has arrived. Once the *in surface* has arrived, a local device can use the push send activity to send data to the remote device.

Detail:
```
AMEVENT psInArrivedEvent(SHANDLE hSession)
```

Relevant files: am_ps.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
tops_am_job.h
tops_oip.h
tops_oiplib.h

Event Sturcture Fields: The field SHANDLE hSession is the session handle used to identify the remote device.

Usage: After a session has been established, the local sending device will receive a psInArrivedEvent after it initializes its push send activity and the remote receiving device initializes its push receive activity. The local sender will receive no notification if the remote device has no push receive activity initialized. This event will not occur after a psShutdown on the local sending side or a prShutdown on the remote receiving side. The event will be posted to the AM event queue and should be retrieved with amGetNextEvent. No push send routines can send information to a device until its session handle arrives in a psInArrivedEvent. A device can continue to use a remote device's *in surface* until a psInGoneEvent arrives for the corresponding session handle.

Requirements: A session must be established, the remote device must have its push receive activity initialized and the local device must have its push send activity initialized.

See also: amStartSession
amSessionStartEvent
psInitialize
psShutdown
psInGoneEvent
prInitialize
prShutdown

Samples:
```
AMEVENT *tempEvent=NULL;
SHANDLE hSession;

amGetNextEvent(&tempEvent);
if (tempEvent->iEvent == psInArrivedEvent)
    hSession = tempEvent->data.psInArrivedEvent.hSession;
```

Identity: psInGoneEvent

Description: This event notifies the local device that the *in surface* of a remote device has been removed.

Detail:
```
AMEVENT psInGoneEvent(SHANDLE hSession)
```

Relevant files: am_ps.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
tops_am_job.h
tops_oip.h
tops_oiplib.h

Event Structure Fields: The field SHANDLE hSession is the session handle used to identify the remote device.

Usage: This event notifies the device that the in surface of the appliance identified by hSession has been removed. This occurs when the receiver shuts down its push receive activity or the session ends. The session handle will correspond to a session handle from a previous psInArrivedEvent.

If the local sending device has established a session with a remote receiving device and received its *in surface*, the remote device may call amEndSession. This will result in the local device receiving a psInGoneEvent in addition to the amSessionEndEvent. Alternatively, if the remote device had called prShutdown instead of amEndSession, the local device would have received just the psInGoneEvent.

Requirements: A psInGoneEvent will only occur after the device receives a psInArrivedEvent and before the device shuts down its push send activity via psShutdown or amShutdown.

See also: amEndSession
amSessionEndEvent
psInArrivedEvent
psShutdown
prShutdown

Samples:

Identity: psStartJob

Description: This routine is used by a local sending device to start a new push job. It allows the remote receiving device to make requests for children and content from the top-level surface created by the psStartJob routine. It is the remote receiving device which controls the transfer of information by requesting specific job elements from the local sender. The sender simply describes everything which is available, then the receiver requests only what it wants.

Detail:

```
DEVAPI Tstatus psStartJob(SNAME nDoc, BD *emDescription, SHANDLE
hTarget, SHANDLE *pDoc, JOBID *pJob)
```

Relevant files: am_ps.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

tops_am_job.h

tops_oip.h

tops_oiplib.h

Parameters :
Return values:

The parameter SNAME nDoc is the text description or name of the surface, which is chosen by the device. Surface names should be unique for any element of this and other jobs that the sender is dealing with at any one time.

The parameter BD *emDescription is the e-material description of the element and its child surfaces.

The parameter SHANDLE hTarget is the session handle of the target device. This value should correspond to the session handle returned in a psInArrivedEvent.

The parameter SHANDLE *pDoc is a unique identifier generated by the JetSend protocol stack to keep track of the top level surface created for this job. This will be used to identify re-requests for the top level surface and requests for content from the top-level surface.

The parameter JOBID *pJob is a unique identifier generated by the JetSend protocol stack to keep track of this job.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: This routine will be used after a session has been established between a sender and receiver and the sender has received a psInArrivedEvent. The session handle returned in the psInArrivedEvent will be used as the hTarget parameter in the psStartJob call. A psStartJob lets the receiver know what information is available by way of a prJobStartEvent on the receiver's side. Based on this event, the receiver may then request content and/or children for the top level surface. The sender will be notified of these requests via the psSurfaceRequestEvent and psContentRequestEvent.

Requirements: After a sending device receives a psInArrivedEvent from a remote receiver, the local sender may call psStartJob until it receives a psInGoneEvent for the remote device or until the local device calls psShutdown.

See also:

psInitialize

psShutdown

psInArrivedEvent

psInGoneEvent

prJobStartEvent

Samples:

```
AMEVENT *tempEvent=NULL;
SHANDLE hTarget, pDoc;
JOBID pJob;
BD *emDescription = buildTestDescription();

amGetNextEvent(&tempEvent);
if (tempEvent->iEvent == psInArrivedEvent)
{
    hTarget = tempEvent->data.psInArrivedEvent.hSession;
    psStartJob("Surface 001", emDescription, hSession, &pDoc,
&pJob);
}
```

Identity: psCancelJob

Description: This routine is used by a sending device to a cancel a job it has started.

Detail:
```
DEVAPI Tstatus psCancelJob(JOBID iJob)
```

Relevant files: am_ps.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
tops_am_job.h
tops_oip.h
tops_oiplib.h

Parameters: The parameter JOBID iJob is the unique identifier generated by the JetSend protocol stack to keep track of this job. It was returned to the device as the pJob parameter in the preceeding psStartJob call.

Return values: This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: This routine may be used by a sending device to cancel a job, if necessary. This will free up the resources associated with that job. The function psCancelJob does not result in a psJobEndEvent on the local sending device. This call will result in a prJobEndEvent on the remote receiving device.

Requirements: The function psCancelJob may only be called with a job id which was returned to the device as a parameter in a psStartJob call.

See also: psStartJob
prJobEndEvent

Samples:
```
JOBID iJob;

psStartJob("Surface 001", emDescription, hSession, &iDoc, &iJob);
...
psCancelJob(iJob);
```

Identity: `psJobEndEvent`

Description: This event notifies the local sending device that the remote receiving device has finished with this job, allowing the local sending device to free up resources associated with the job.

Detail:
```
AMEVENT psJobEndEvent(JOBID iJob)
```

Relevant files: am_ps.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
tops_am_job.h
tops_oip.h
tops_oiplib.h

Event Structure Fields: The field JOBID iJob is a unique identifier generated by the JetSend protocol stack to keep track of this job. It was returned to the device as the pJob parameter in the preceeding psStartJob call.

Usage: When a remote receiver is done with a job (it has requested all desired surfaces and/or content), it will call prEndJob. The local sending device will be notified of this via a psJobEndEvent. Information about the final outcome of the job may be available through the job status activity, if the receiver has implemented status. The receiver may end the job before the job is actually completed (ie a printer may end the job after it has received the entire image, but before it has been processed and printed), so in this case, the sender could only determine the final outcome of the job via the status activity.

Requirements: A psJobEndEvent(iJob) will only occur after a successfull call to psStartJob(..., &iJob). A psJobEndEvent will not occur when the local sender calls psCancelJob, but will occur when the remote receiver calls prEndJob.

See also: psStartJob
prEndJob

Samples:
```
psInArrivedEvent();
psStartJob();
/* Handle  psSurfaceRequestEvent*/
 /* Handle psContentRequestEvent*/
psSendSurface();
psSendContent();
...
 /* Handle psSurfaceReleasedEvent */
 /* Handle psJobEndEvent*/
```

Identity: psSessionFromJob

Description: This returns the session handle associated with a given job, allowing the device to associate a job with a remote device.

Detail:
```
DEVAPI Tstatus psSessionFromJob(JOBID iJob, SHANDLE *hSession)
```

Relevant files: am_ps.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
tops_am_job.h
tops_oip.h
tops_oiplib.h

Parameters:
Return values: The parameter JOBID iJob is a unique identifier generated by the JetSend protocol stack to keep track of a job. iJob was returned to the device as the pJob parameter in a preceeding psStartJob call.

The parameter SHANDLE *hSession is the session handle used to identify the remote device, and is returned as a parameter of this function.. This session handle corresponds to the hTarget parameter used in the psStartJobCall which corresponds to a iJob.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: None.

Requirements: This function can only be called after a psStartJob and before a psJobEndEvent occur for the given job id.

See also: psJobStartEvent
psJobEndEvent

Samples: None.

Identity: psSurfaceRequestEvent

Description: This event notifies the local sending device of a request for the description of a surface. The remote receiving device controls the transfer of information by issuing requests for the description of surfaces and content which the local sender responds to. This event is one which notifies the sender of a receiver's request.

Detail:
```
AMEVENT psSurfaceRequestEvent(SNAME nElement, REQID iRequestid)
```

Relevant files: am_ps.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
tops_am_job.h
tops_oip.h
tops_oiplib.h

Event Structure Fields: The field SNAME nElement is the textual name or description of the surface being requested.

The field REQID iRequestid is a unique identifier generated by the JetSend protocol stack to keep track of this request. This id should be used when fullfilling the request via psSendSurface.

Usage: After the local sender has started a job via psStartJob, the sending device may receive one or more psSurfaceRequestEvents. This request could be for any information related to this job. It could be either a re-request for the description of the top level surface or a request for the description of a child of the top level surface. After a psSurfaceRequestEvent, the device should either call psSendSurface or psRejectSurfaceRequest to fullfill or reject the request, respectively.

Requirements: This event is generated by a remote receiving device making a prRequestSurface call. This can occur as a result of a successful psStartJob or a psSendSurface call.

See also: psStartJob
psSendSurface
psSurfaceReleasedEvent
psRejectSurfaceRequest
prRequestSurface

Samples:
```
psStartJob();
/* Handle psSurfaceRequestEvent*/
psSendSurface();
```

Identity: psSendSurface

Description: This routine is used by a local sender to send a surface description to a remote receiving device. The surface is sent in response to a request for the surface by the remote receiving device.

Detail:
```
DEVAPI Tstatus psSendSurface(REQID iRequestId, BD *emReply,
SHANDLE *pElement)
```

Relevant files: am_ps.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

tops_am_job.h

tops_oip.h

tops_oiplib.h

Parameters:
Return values: The parameter REQID iRequestId is used to identifiy which request is being satisified. This corresponds to the iRequestId returned in a preceeding psSurfaceRequestEvent.

The parameter BD *emReply holds a description of the surface, the format of which is defined by the E-material specification. This buffer will be freed by the JetSend protocol stack when it is no longer needed.

The parameter SHANDLE *pElement is a unique identifier generated by the JetSend protocol stack to keep track of the surface created for this send. This will correspond with the hElement which will be returned in a subsequent psSurfaceReleasedEvent.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: See requirements.

Requirements: This function should only be called in response to a psSurfaceRequestEvent. The resources associated with the sent surface should not be released until the device receives a psSurfaceReleasedEvent. The BD *emReply should never be freed by the device, as it is freed by the JetSend protocol stack.

See also: psStartJob

psSurfaceRequestEvent

psSurfaceReleasedEvent

psRejectSurfaceRequest

Samples:
```
psStartJob();
/* Handle psSurfaceRequestEvent */
psSendSurface();
```

Identity: psSurfaceReleasedEvent

Description: This event notifies the local sending device that the resources associated with a given surface can be released. This means that the local sender will not get any further content requests for this surface unless its description is requested again.

Detail:
```
AMEVENT psSurfaceReleasedEvent(SHANDLE hElement)
```

Relevant files: am_ps.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
tops_am_job.h
tops_oip.h
tops_oiplib.h

Event Structure Fields: The field SHANDLE hElement is the handle associated with the surface which as been released. This handle corresponds the the surface handle returned from both the psStartJob and psSendSurface calls.

Usage: A psSurfaceReleasedEvent will eventually occur in response to each psSendSurface done by the local sending device to let the local sender know when it can free the resources associated with the sent surface.

Requirements: This can only occur after a successful psSendSurface by the local sender and either the remote receiving device calls prReleaseSurface or the session ends.

See also: psStartJob
psSurfaceRequestEvent
psSendSurface
psSurfaceReleasedEvent

Samples:
```
psInArrivedEvent();
psStartJob();
/* Handle psSurfaceRequestEvent*/
/* Handle psContentRequestEvent*/
psSendSurface();
psSendContent();
...
/* Handle psSurfaceReleasedEvent*/
/* Handle psJobEndEvent*/
```

Identity: psRejectSurfaceRequest

Description: This routine allows the local sender to reject the remote receiver's request for a surface if the sender is unable to satisfy the request.

Detail:
```
DEVAPI Tstatus psRejectSurfaceRequest(REQID iRequestId)
```

Relevant files: am_ps.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
tops_am_job.h
tops_oip.h
tops_oiplib.h

Parameters : The parameter REQID iRequestId is used to identifiy which request is being rejected. This corresponds to the iRequestId returned in a preceeding psSurfaceRequestEvent.

Return values: This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: The local sending device calls this function if it is unable to satisfy the request for a surface. The remote receiving device may end the job as a result. When the sender calls psRejectSurface, the remote receiver will get a prSurfaceEvent with a surface handle of zero, indicating rejection the local sender rejected its request.

Requirements: This function may only be called in response to a psSurfaceRequestEvent with a corresponding request id.

See also: psStartJob
psSurfacRequestEvent
psSendSurface
psSurfaceReleasedEvent
prEndJob

Samples:
```
psStartJob();
/* Handle psSurfaceRequestEvent */
psRejectSurfaceRequest();
```

Identity: psContentRequestEvent

Description: This event notifies the local sending device of a request for content by a remote receiving device.

Detail:
```
AMEVENT psContentRequestEvent(SHANDLE hElement, BD *emRequest,
Tu16 iMaxBufferSize, REQID iRequestId)
```

Relevant files: am_ps.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

tops_am_job.h

tops_oip.h

tops_oiplib.h

Event Structure Fields: The field SHANDLE hElement specifies the surface containing the desired content. This surface will either be the top-level surface sent in a previous psStartJob call, or a lower level surface sent via psSendSurface.

The field BD *emRequest specifies what part and what encoding of the content the target requires. The exact format of the request block is defined in the E-material specification. The local sending device should free this buffer when it is no longer needed, or call amFreeEvent to free the entire event.

The field Tu16 iMaxBufferSize denotes the maximum buffer size that the sending device can send using psSendContent for this request. If the device has more data than the buffer size allows, it should split the data into separate blocks and call psSendContent multiple times with the same request id. See psSendContent for more info.

The field REQID iRequestId is the identifier for this request. This value should be used in the response, whether it is psSendContent or psRejectContentRequest. If the content is too large to fit in a single response, multiple psSendContent calls with this request id will be made by the local sending device.

Usage: After a job has been started by the local sending device and 0 or more additional surfaces have been sent, the remote receiving device will know which information is available. The remote receiver will then request the information it wants. The local sender will be notified of the request via a psContentRequestEvent. In response to this request, the local sender should either call psSendContent with the requested data, or call psRejectContentRequest.

Requirements: This event can only occur on a local sending device while a job is in progress (after a psStartJob and before a psJobEndEvent).

See also: psStartJob

psSendSurface

psSendContent

psRejectContentRequest

psJobEndEvent

Samples:

```
/* Handle psInArrivedEvent */
psStartJob();
/* Handle psContentRequestEvent This is a request for content
from the top level description. */
/* Handle psSurfaceRequestEvent */
psSendContent();
psSendSurface();
/* Handle psContentRequestEvent. This is a request for content
from the 2nd description sent. */
psSendContent();
/* Handle psJobEndEvent */
```

Identity: psSendContent

Description: This routine is used by a local sending device to send requested content data to a remote receiving device.

Detail:
```
DEVAPI Tstatus psSendContent(REQID iRequestId, BD *pData, Tbool
bLastBlock)
```

Relevant files: am_ps.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

tops_am_job.h

tops_oip.h

tops_oiplib.h

Parameters :
Return values:

The parameter REQID iRequestId is the request identifier from the psContentRequestEvent to which this call is a response.

The parameter BD *pData is a buffer containing the content data to be sent. This buffer will be freed by the JetSend protocol stack when it has been successfully sent. The size of this buffer is limited to iMaxBufferSize, a value returned in the preceeding psContentRequestEvent. The entire buffer will be sent to the remote receiving device regardless of the amount of data in the buffer. Thus, the local sending device should allocate the smallest buffer possible which will contain the content data. If the size of the requested content is greater than iMaxBufferSize, the local sender should call psSendContent serially multiple times with the same iRequestId, with the content data split into the buffers sent with each call.

The parameter Tbool bLastBlock specifies whether or not this is the last block of content data. It should be set to true(1) if this is the last block, or only block, sent to satisfy a given content block. If this isn't the last or only block being sent, it should be false(0). The latter will only occur if the size of the content data is greater than iMaxBufferSize and the content data is being sent via multiple calls to psSendContent.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: This routine is one of two routines which may be called in response to a psContentRequestEvent. If the content data is too large to fit in a single buffer of iMaxBufferSize (returned in the psContentRequestEvent), the local sending device should split the content into multiple buffers and call psSendContent multiple times with the same requestId. In this case, bLastBlock should be set to false(0) for all but the last block of the content data. This routine will result in a prContentEvent on the remote receiving device.

Requirements: This routine will only be called by a local sending device in response to a psContentRequestEvent.

See also:

psStartJob
psSendSurface
psContentRequestEvent
psJobEndEvent
prContentEvent

Samples:

```
/* Handle psInArrivedEvent */
psStartJob;
/* Handle psSurfaceRequestEvent */
psSendSurface();
/* Handle psContentRequestEvent */
psSendContent();
/* Handle psJobEndEvent */
```

Identity: psRejectContentRequest

Description: If a local sending device is unable to satisfy a content request, it should call this function with the request id of the content request. This informs the remote receiving device no content data will be arriving for this request.

Detail:
```
DEVAPI Tstatus psRejectContentRequest(REQID iRequestId)
```

Relevant files: am_ps.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
tops_am_job.h
tops_oip.h
tops_oiplib.h

Parameters: The parameter REQID iRequestId is the request identifier from the psContentRequestEvent which is being rejected.

Return values: This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: If a local sending device is unable to satisfy a request for content, it should call this function. This will result in a prContentRejectedEvent on the remote receiving device. The remote receiver may end the job as a result of receiving this event.

Requirements: This routine will only be called by a local sending device in response to a psContentRequestEvent.

See also: psStartJob
psSendSurface
psContentRequestEvent
psJobEndEvent
prContentRejectedEvent

Samples:
```
/* Handle psInArrivedEvent */
psStartJob();
/* Handle psContentRequestEvent */
psRejectContentRequest();
/* Handle psJobEndEvent */
```

Identity: psDescriptionRequestEvent

Description: This event tells the local sending device that a remote receiving device has requested part of a description. This will only occur if the initial description was too large to fit in a single description block (rare).

Detail:
```
AMEVENT psDescriptionRequestEvent(SHANDLE hElement, BD
*emRequest, REQID iRequestId)
```

Relevant files: am_ps.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

tops_am_job.h

tops_oip.h

tops_oiplib.h

Event Sturcture Fields: The field SHANDLE hElement specifies the surface containing the desired description. This surface will either be the top-level surface sent via a psStartJob, or a lower level surface sent via psSendSurface.

the field BD *emRequest specifies what part of the description the target (the remote receiving device) requires. The exact format of the request block is defined in the E-material Reference. The device should free this buffer when it is no longer needed, or call amFreeEvent to free the entire event.

The field REQID iRequestId is the identifier for this request. This value should be used in the subsequent psSendDescription call made in response to this event.

Usage: When a remote receiving device receives a partial E-material description (because the size of the description is greater than AM_MaxDescriptionSize), it will use prRequestDescription to request additional parts of the description. See the E-material Reference for guidelines. This call on the remote receiver results in a psDescriptionRequestEvent on the local sending device. The local sender, in response to this event, will call psSendDescription to send the requested information. If the entire description was sent initially (via psStartJob or psSendSurface), the local sender is not required to respond to psDescriptionRequestEvents for that description.

Requirements: This event can only occur on a local sending device while a job is in progress.

See also: psSendDescription

prRequestDescription

prDescriptionEvent

psStartJob

psSendSurface

Samples:

```
amInitialize();
psInitialize();
amStartSession();
/* Handle amSessionStartEvent */
psStartJob();
/* Handle psDescriptionRequestEvent */
psSendDescription();
```

Identity: psSendDescription

Description: This function is used by a local sending device to satisfy a request for a description.

Detail:
```
DEVAPI Tstatus psSendDescription(SHANDLE hElement, BD
*emDescription)
```

Relevant files: am_ps.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

tops_am_job.h

tops_oip.h

tops_oiplib.h

Parameters:
Return values: The parameter BD *emReply holds a description of the surface, the format of which is defined by the E-material Reference. This buffer will be freed by the JetSend protocol stack when it is no longer needed.

The parameter REQID iRequestId is the identifier for this request. This value corresponds to the iRequestId passed up in the psDescriptionRequestEvent to which this function is replying.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: When a remote receiving device receives a partial E-material description (because the size of the description is greater than AM_MaxDescriptionSize), it will use prRequestDescription to request additional parts of the description. See the E-material Reference for guidelines. This call on the remote receiver results in a psDescriptionRequestEvent on the local sending device. The local sender, in response to this event, will call psSendDescription to send the requested information. If the entire description was sent initially (via psStartJob or psSendSurface), the local sender is not required to respond to psDescriptionRequestEvents for that description.

Requirements: This function should only be called by a sending device in response to a psDescriptionRequestEvent.

See also: psDescriptionRequestEvent

prRequestDescription

prDescriptionEvent

psStartJob

psSendSurface

Samples:

```
amInitialize();
psInitialize();
amStartSession();
/* Handle amSessionStartEvent */
psStartJob()
/* handle psDescriptionRequestEvent */
psSendDescription()
```

Identity: prInitialize

Description: This routine initializes the receiver side of the push activity, which is used by receiving devices to receive E-material.

Detail:
```
DEVAPI Tstatus prInitialize(void)
```

Relevant files: am_pr.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
tops_am_job.h
tops_oip.h
tops_oiplib.h

Parameters: This routine takes no arguments.

Return values: This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: Any receiving device wishing to implement the receiver side of the push activity must call this routine. It will often be called immediately following amInitialize. It must return successfully before any other push receive activity routines may be called. When a session is established with a remote sending device, the local receiver's in surface will automatically be sent on the sender, resulting in a psInArrivedEvent on the remote sending device. This allows the remote sender to start push jobs (begin the exchange of info).

Requirements: This routine may only be called by a receiving device after a successful amInitialize and before an amShutdown. After prInitialize returns successfully, it should not be called again unless there has been a prShutdown.

See also: amInitialize
amShutdown
prShutdown
psInArrivedEvent
prJobStartEvent

Samples:

```
amInitialize();
prInitialize();
/* Handle amSessionStartEvent A sender will usually initiate the
session. */
/* Handle prJobStartEvent*/
prRequestSurface();
/* Handle prSurfaceEvent */
prRequestContent()
/* Handle prContentEvent */
/* Handle prContentEvent. More than one may occur for a single
request. */
prEndJob();
amEndSession()
/* Both the receiver and sender can end the session,
regardless of which started the session. */
/* Handle amSessionEndEvent */
```

Identity: prShutdown

Description: This routine is used by a receiving device to shut down the push receiver activity and free any associated resources.

Detail:
```
DEVAPI Tstatus prShutdown(void)
```

Relevant files: am_pr.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
tops_am_job.h
tops_oip.h
tops_oiplib.h

Parameters: This routine takes no arguments.

Return values: This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: This routine will be called when a receiving device wants to shut down the push receive activity and free its resources. Calling this routine will result in a psInGoneEvent on the remote sending device, if a session is established.

Requirements: This routine may only be called by a receiving device after a successful amInitialize and before an amShutdown. prInitialize must also return successfully before this routine may be called.

See also: amInitialize
amShutdown
prInitialize
psInGoneEvent

Samples:
```
amInitialize(0);
prInitialize();
...
prShutdown();
amShutdown();
```

Identity: prInRejectedEvent

Description: This event notifies the local receiving device that its *in surface* has been unimpressed by remote sending device. This means the remote sender won't satisfy any [further] requests for descriptions or content.

Detail:
```
AMEVENT prInRejectedEvent(SHANDLE hSession)
```

Relevant files: am_pr.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
tops_am_job.h
tops_oip.h
tops_oiplib.h

Event Structure Fields: The field SHANDLE hSession is the session handle identifying the remote sending device which has unimpressed the local receiver's *in surface*.

Usage: This event notifies the local receiving device that its *in surface* has been unimpressed by the remote sending device. This means that the remote device does not have a push send module installed, so the local device will never receive anything from it.

Requirements: This event will only occur after a session has been established. At this point, the event may be caused by one of three things: the remote device has no push send module initialize; the remote device shuts down its push send module; or the session has ended.

See also: amStartSession
amEndSession
prInitialize
psInitialize
psShutdown

Samples:
```
amInitialize();
prInitialize();
amStartSession();
/* Handle amSessionStartEvent */
/* Handle prInRejectedEvent */
```

Identity: prJobStartEvent

Description: This event notifies the local receiving device that a remote sending device has pushed some E-material to the local receiver's *in surface*. In this case, this E-material is on the top-level surface of a job.

Detail:
```
AMEVENT prJobStartEvent(SNAME nDoc, JOBID iJob, SHANDLE hDoc, BD
*emDescription)
```

Relevant files: am_pr.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

tops_am_job.h

tops_oip.h

tops_oiplib.h

Event Structure Fields: The field SNAME nDoc is the textual description, or name, of the surface which has arrived.

The field JOBID iJob uniquely identifies the job for the local receiving device. This job id is used to identify the job in prEndJob, prJobEndEvent and prSessionFromJob.

The field SHANDLE hDoc is the sending device's identifer for the top-level surface of the job This handle is used to identify the surface in prReleaseSurface, prSurfaceReleasedEvent, and prRequestContent.

The field BD *emDescription holds a description of the surface, the format of which is defined by the E-material Reference. The local receiving device should free this buffer when no longer needed, or call amFreeEvent to free the entire event.

Usage: This event notifies the local receiver that a remote sending device has started a job. From the top-level surface which is impressed on the local receiver's *in surface*, the receiver may then make requests for child surfaces or for content of the top-level surface.

Requirements: This event may only occur on a receiving device after prInitialize returns successfully and before a prShutdown, and after a session has been established with a remote sending device.

See also: prInitialize

prShutdown

amStartSession

amSessinStartEvent

psInitialize

Samples:

```
amInitialize();
prInitialize();
amSessionStartEvent     /* Handle ---------- */
prJobStartEvent         /* Handle ---------- */
prRequestSurface();
prSurfaceEvent          /* Handle ---------- */
prRequestContent();
prContentEvent          /* Handle ---------- */
prReleaseSurface
prEndJob();
```

Identity: prEndJob

Description: This routine allows a local receiving device to notify a remote sending device that a job is finished.

Detail:
```
DEVAPI Tstatus prEndJob(JOBID iJob)
```

Relevant files: am_pr.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
tops_am_job.h
tops_oip.h
tops_oiplib.h

Parameters: The parameter JOBID iJob uniquely identifies the job for the local receiving device. This value should be the iJob returned in a previous prJobStartEvent for the corresponding job.

Return values: This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: This routine allows the local receiver to end or cancel a job. This results in a psJobEndEvent on the remote sending device. The local receiver can indicate in the job status whether the job was successfully processed or not. The call to this function does not result in a prJobEndEvent on the local receiver (a prJobEndEvent only occurs on the local receiver when the job is cancelled by the remote sender).

Requirements: This function can only be called with a job id corresponding to a job which has been started (this value is returned in the prJobStartEvent). Additionally, this function can only be called before a prJobEndEvent, which signifies the remote sending device has ended the job first via psCancelJob.

See also: prJobStartEvent
psJobEndEvent
psCancelJob

Samples:
```
amInitialize();
prInitialize();
/* Handle amSessionStartEvent */
prJobStartEvent */
prRequestSurface();
/* Handle prSurfaceEvent */
prRequestContent();
/* Handle prContentEvent */
prReleaseSurface();
prEndJob();
```

Identity: prJobEndEvent

Description: This event notifies the local receiving device that a remote sending device has cancelled a job.

Detail:
```
AMEVENT prJobEndEvent(JOBID iJob)
```

Relevant files: am_pr.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
tops_am_job.h
tops_oip.h
tops_oiplib.h

Event Structure Fields : The field JOBID iJob uniquely identifies the job for the local receiving device. This value corresponds to the iJob returned in a previous prJobStartEvent.

Usage: See reqirements.

Requirements: This event can only occurnon a receving device after a corresponding prJobStartEvent, and after a remote sending device cancels the job.

See also: prJobStartEvent
psCancelJob

Samples:
```
amInitialize();
prInitialize();
/* Handle amSessionStartEvent */
/* Handle prJobStartEvent */
prRequestContent();
/* Handle prContentEvent */
/* Handle prJobEndEvent */
```

Identity: prSessionFromJob

Description: This returns the session handle associated with a given job id.

Detail:
```
DEVAPI Tstatus prSessionFromJob(JOBID iJob, SHANDLE *hSession)
```

Relevant files: am_pr.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
tops_am_job.h
tops_oip.h
tops_oiplib.h

Parameters:
Return values: The parameter JOBID iJob uniquely identifies the job for the local receiving device. This value corresponds to the iJob returned in the preceeding prJobStartEvent for this job.

The parameter SHANDLE *hSession is the session handle used to identify the remote sending device, which is returned as a parameter of this function. This session handle corresponds to the hSession parameter returned in the amSessionStartEvent for the remote receiving device.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: The local receiving device may call this function to determine which job originates from which device.

Requirements: This function may only be called by the local receiving device while a job is in progress (after a job has started and before that job has ended).

See also: Handle prJobStartEvent
Handle prJobEndEvent
prEndJob

Samples: None.

Identity: prRequestSurface

Description: This routine allows a local receiving device to request the a surface containing an E-material description from a remote sending device.

Detail:
```
DEVAPI Tstatus prRequestSurface(SHANDLE hSession, SNAME nElement,
REQID *pRequestId)
```

Relevant files: am_pr.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

tops_am_job.h

tops_oip.h

tops_oiplib.h

Parameters :
Return values: SHANDLE hSession is the session handle of the remote sending device. This corresponds to the value returned in a previous amSessionStartEvent or from prSessionFromJob.

SNAME nElement is the name of the surface being requested, which was supplied in the description of a previously supplied surface (from a prJobStartEvent or prSurfaceEvent).

REQID *pRequestId is returned as a unique identifier for the request which the local receiving device should use to identify the subsequent prSurfaceEvent.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: After a job has started, the E-material description may contain the name of child surfaces. These child surfaces may be requested by thelocal receiving device via prRequestSurface calls. A prRequestSurface call will result in a prSurfaceEvent, after the remote sending device responds via psSendSurface.

Requirements: The local receiving device must have the name of the surface to be requested. This name will come from the E-material description in a previous prJobStartEvent or a prSurfaceEvent. The local receiver should call prReleaseSurface after it is done with the information from a prSurfaceEvent (this notifies the remote sender it can now free up the resources associated with the surface).

See also: prJobStartEvent

prJobEndEvent

prSurfaceEvent

prReleaseSurface

prSurfaceReleasedEvent

Samples:

```
amInitialize();
prInitialize();
/* Handle amSessionStartEvent */
/* Handle prJobStartEvent */
prRequestSurface();
/* Handle prSurfaceEvent */
prRequestContent();
/* Handle prContentEvent */
prReleaseSurface();
prEndJob();
```

Identity: prSurfaceEvent

Description: This event returns a description that the local receiving device has requested via a prRequestSurface call.

Detail:
```
AMEVENT prSurfaceEvent(REQID iRequestId, SHANDLE hElement, BD
*emDescription)
```

Relevant files: am_pr.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

tops_am_job.h

tops_oip.h

tops_oiplib.h

Event Structure Fields: The field REQID iRequestId is a unique identifier for the request which was returned to the local receiving device in the previous, corresponding prRequestSurface call.

The field SHANDLE hElement identifies the job element that has been received. This corresponds to the hElement used in prReleaseSurface, prSurfaceReleasedEvent and prRequestContent. If hElement is 0, this means that the surface request has been rejected by the remote device.

The field BD *emDescription holds a description of the surface, the format of which is defined by the E-material Reference. The local receiving device should free this buffer when no longer needed, or call amFreeEvent to free the entire event.

Usage: See requirements.

Requirements: A prSurfaceEvent will only arrive as the result of a prRequestSurface call. When a local receiving device is no longer interested in the surface identified by hElement, it should call prReleaseSurface to notify the remote sending device and allow it to free up resources associated with this surface.

See also: prRequestSurface

prReleaseSurface

prSurfaceReleasedEvent

Samples:
```
amInitialize();
prInitialize();
/* Handle amSessionStartEvent */
/* Handle prJobStartEvent */
prRequestSurface();
/* Handle prSurfaceEvent */
prRequestContent();
/* Handle prContentEvent */
prReleaseSurface();
prEndJob();
```

Identity: prRequestDescription

Description: This function is used by the local receiving device to request remaining parts of an E-material description, if the original description was too large for the remote sending device to fit in a single description block.

Detail:
```
DEVAPI Tstatus prRequestDescription(SHANDLE hElement, BD
*emRequest, REQID *pRequestId)
```

Relevant files: am_pr.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

tops_am_job.h

tops_oip.h

tops_oiplib.h

Parameters:
Return values: The parameter SHANDLE hElement is the name of the surface whose description is being requested. This value is either from the prJobStartEvent or a prSurfaceEvent that contained the first part of the description.

The parameter BD *emRequest specifies what part of the description the local receiving device wants. The exact format of the request block is defined in the E-material Reference. The JetSend protocol stack will free this memory when it is no longer needed.

The parameter REQID *pRequestId is returned as a unique identifier for the request which the local receiving device should use to identify the subsequent prDescriptionEvent.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: When the local receiving device receives a partial E-material description (because the size of the description is greater than AM_MaxDescriptionSize), it will use prRequestDescription to request additional parts of the description. See the E-material Reference for guidelines. This call on the local receiver results in a psDescriptionRequestEvent on the remote sending device. The remote sender, in response to this event, will call psSendDescription to send the requested information to the local receiver. If the entire description was received initially, the remote sender is not required to respond to additional prRequestDescriptions for that description.

Requirements: This function should only be called by a local receiving device after a partial description has been received (in a prJobStartEvent or prSurfaceEvent). If the entire description was received in the prJobStartEvent or prSurfaceEvent, the remote sender is not required to respond to re-requests of the same description.

See also:

prJobStartEvent
prSurfaceEvent
prDescriptionEvent
psDescriptionRequestEvent
psSendDescription

Samples:

```
amInitialize();
prInitialize();
/* Handle amSessionStartEvent */
/* Handle prJobStartEvent */
prRequestDescription();
/* Handle prDescriptionEvent */
```

Identity: prDescriptionEvent

Description: This event is a response to a request for a description made by a local receiving device.

Detail:
```
AMEVENT prDescriptionEvent(REQID iRequestId, BD *emDescription)
```

Relevant files: am_pr.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
tops_am_job.h
tops_oip.h
tops_oiplib.h

Event Structure Fields: The field REQID iRequestId is a unique identifier for the request which was returned to the local receiving device in the preceeding prRequestDescription call.

The field BD *emDescription holds the requested description, the format of which is defined by the E-material Reference. The local receiving device should free this buffer when no longer needed, or call amFreeEvent to free the entire event.

Usage: When the local receiving device receives a partial E-material description (because the size of the description is greater than AM_MaxDescriptionSize), it will use prRequestDescription to request additional parts of the description. See the E-material Reference for guidelines. This call on the local receiver results in a psDescriptionRequestEvent on the remote sending device. The remote sender, in response to this event, will call psSendDescription to send the requested information to the local receiver. If the entire description was received initially, the remote sender is not required to respond to additional prRequestDescriptions for that description.

Requirements: This event will only occur on the local receiving device in response to a succesful prRequestDescription call, and only if the original description was an incomplete description.

See also: prRequestDescription
psDescriptionRequestEvent
psSendDescription

Samples:
```
amInitialize();
prInitialize();
/* Handle amSessionStartEvent */
/* Handle prJobStartEvent */
prRequestDescription();
/* Handle prDescriptionEvent */
```

Identity: prReleaseSurface

Description: This routine notifies the remote sending device that the local receiving device is no longer interested in a specific surface. This allows the remote sender to free up resources associated with that surface.

Detail:
```
DEVAPI Tstatus prReleaseSurface(SHANDLE hElement)
```

Relevant files: am_pr.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
tops_am_job.h
tops_oip.h
tops_oiplib.h

Parameters: The parameter SHANDLE hElement is the identifier of the surface to be released. It corresponds to the hElement in the prSurfaceEvent.

Return values: This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: This function should be used when a local receiving device is done with a surface which has arrived via a prSurfaceEvent. This call will result in a psSurfaceReleasedEvent on the remote sending device. It will not result in a prSurfaceReleasedEvent on the local receiver. If the receiver uses this function on the top-level surface that constitues the job, it will be equivalent to a prEndJob and the job will end. After releasing a surface, the local receiver cannot make any further content requests for the surface unless it issues another prRequestSurface and receives another prSurfaceEvent first.

Requirements: This function should be used when a local receiving device is done with a surface which has arrived via a prSurfaceEvent. All surfaces a local receiver receives should be released at some point before the job ends.

See also: prJobStartEvent
prEndjob
prRequestSurface
prSurfaceEvent
psSurfaceReleasedEvent
prSurfaceReleasedEvent

Samples:

```
amInitialize();
prInitialize();
/* Handle amSessionStartEvent */
/* Handle prJobStartEvent */
prRequestSurface();
/* Handle prSurfaceEvent */
prRequestContent();
 /* Handle prContentEvent */
prReleaseSurface();
prEndJob()
```

Identity: prSurfaceReleasedEvent

Description: This event notifies the local receiving device that a remote sending device has freed the resources associated with this surface and no more content requests can be issued for this surface.

Detail:
```
AMEVENT prSurfaceReleasedEvent(SHANDLE hElement)
```

Relevant files: am_pr.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
tops_am_job.h
tops_oip.h
tops_oiplib.h

Event Structure Fields: The field SHANDLE hElement is the identifier of the surface which has been released. It corresponds to the hElement in the prSurfaceEvent.

Usage: This event can only occur for a surface which has been requested and received by the local receiving device via prRequestSurface and prSurfaceEvent events. This event is the result of the session ending or the remote sending device shutting down its push send activity. There is no call which allows the remote sendet to explicitly release a surface.

Requirements: This event can only occur on a local receiving device for a surface which has been requested and received via prRequestSurface and prSurfaceEvent. Once this occurs, no more content requests can be made for this surface. The surface may be re-requested, but it is likely that the remote sending device will reject these requests if it has freed the resources associated with the surface.

See also: prRequestSurface
prSurfaceEvent

Samples: None.

Identity: prRequestContent

Description: This routine requests the transfer of the content of a given surface. The receiver controls the transfer of information by issuing requests for the content of surfaces for which it has previously received a descriptoin.

Detail:
```
prRequestContent(SHANDLE hElement, BD *emRequest, Tu8
iChannelType, REQID *pRequestId)
```

Relevant files: am_pr.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

tops_am_job.h

tops_oip.h

tops_oiplib.h

Parameters :
Return values:

The parameter SHANDLE hElement is the name of the surface whose content is being requested. This name is either from the prJobStartEvent or a prSurfaceEvent.

The parameter BD *emRequest specifies how the content is required and what part of the content is required. The format for this request is defined by the e-material specification. The infrastructure will free emRequest when it is no longer needed.

The parameter Tu8 iChannelType denotes the type of tranport that should be used for the requested data. Currently, there are three possible values for this: AM_CHANNEL_MESSAGE, AM_CHANNEL_DONTCARE, and AM_CHANNEL_STREAM.

The parameter REQID *pRequestId is returned to identify the subsequent incoming content. It corresponds to the iRequestId paramter in prContentEvent and prContentRejectedEvent.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See API Reference Overview for more information.

Usage: After the receiving device has received a description surface, it will use prRequestContent to request the content of that surface. This will result in either one or more prContentEvents or a prContentRejectedEvent.

Requirements: The receiver must have first received a surface structure before content can be requested. This surface structure can be the top-level surface which arrives in a prJobStartEvent or it can be a surface structure which arrive as a result of subsequent prRequestSurface calls.

See also: prSurfaceEvent

prRequestSurface

prContentEvent

prContentRejectedEvent

Samples:

```
amInitialize();
prInitialize();
/* Handle amSessionStartEvent */
/* Handle prJobStartEvent */
prRequestContent();
/* Handle prContentEvent */
prRequestSurface();
/* Handle prSurfaceEvent */
prRequestContent();
/* Handle prContentEvent */
prReleaseSurface();
prEndJob();
```

Identity: prContentEvent

Description: This routine notifies the local receiving device that a previously requested piece of content has arrived from the remote sending device.

Detail:
```
AMEVENT prContentEvent(REQID iRequestId, BD *pData, Tbool
bLastBlock)
```

Relevant files: am_pr.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

tops_am_job.h

tops_oip.h

tops_oiplib.h

Event Structure Fields: The field REQID iRequestId is the unique identifier of the request. It matches the request id returned in an earlier call to prRequestContent.

The field BD *pData is the buffer containing the requested information. The local receiving device should free this memory when it is no longer needed, or call amFreeEvent to free the entire event.

The field Tbool bLastBlock is true (1) when there will be no more content events coming for a given request. Otherwise it will be false (0), in which case the device should expect more prContentEvents with the same request id. This indicates the size of the content block was too large to fit in a single content data buffer.

Usage: This routine notifies the local receiving device that a previously requested piece of content has arrived from the remote sending device. The amount of data the remote ender can send in one buffer is limited to the iMaxBufferSize it receives in the psContentRequestEvent. This means that some data may not fit in a single buffer. In this case, the remote sender will call psSendContent multiple times with the same request id, where each buffer contains part of the requested content. Each psSendContent will result in a prContentEvent on the local receiver. The last block of content data will have the bLastBlock value set to true. Multiple content blocks will arrive serially. If all the the content data fit in a single buffer, the local receiver will get a single prContentEvent, with bLastBlock set to true.

Requirements: This event will only occur on a local receiving device in response to a prRequestContent call. After this event occurs, it is the responsibility of the local receiving device to free the memory in pData, or call amFreeEvent to free this memory.

See also: prRequestContent

prContentRejectedEvent

psSendContent

psRejecteContentRequest

Samples:

```
amInitialize();
prInitialize();
/* Handle amSessionStartEvent */
/* Handle prJobStartEvent */
prRequestContent();
/* Handle prContentEvent */
prEndJob();
```

Appendix F

Status Send/Status Receive Activity

Part of the Device Independent API

Identity: ssInitialize

Description: This routine initializes the sender side of the status activity.

Detail:
```
DEVAPI Tstatus ssInitialize(void)
```

Relevant files: am_ss.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
appl_em.h
appl_emlib.h
tops_am_job.h

Parameters: This routine takes no arguments.

Return values: This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h discussion in Chapter 1 for more information.

Usage: This routine initializes the sender side of the status activity. This will normally be on a receiving device, such as a printer, to allow it to send status information.

Requirements: This routine may only be called by a status sender after a successful amInitialize and before an amShutdown. After ssInitialize returns successfully, it should not be called again unless there has been an ssShutdown. If the status sender is localized, ssSetLanguages should be called after ssInitialize.

See also: amInitialize
amShutdown
ssShutdown
ssSetLanguages
Localization

Samples:

```
amInitialize();
ssInitialize();
```

Identity: ssShutdown

Description: This routine shuts down the sending side of the status activity and frees any associates resources.

Detail:
```
DEVAPI Tstatus ssShutdown(void)
```

Relevant files: am_ss.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
appl_em.h
appl_emlib.h
tops_am_job.h

Parameters: This routine takes no arguments.

Return values: This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h discussion in Chapter 1 for more information.

Usage: This routine shuts down the sending side of the status activity.

Requirements: This routine may only be called by a status sender after a successful amInitialize and before an amShutdown. ssInitialize must also return successfully before this routine may be called.

See also: amInitialize
amShutdown
ssInitialize

Samples:
```
amInitialize();
ssInitialize();
ssShutdown();
amShutdown();
```

Identity: ssSetLanguages

Description: This function should be called by localized status senders after the status module has been initialized to define a list of languages that the device will provide all its status messages in.

Detail:
```
DEVAPI Tstatus ssSetLanguages(Tu16 iLanguages, LOCLANG
*pLanguages)
```

Relevant files: am_ss.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

appl_em.h

appl_emlib.h

tops_am_job.h

Parameters:
Return values:

The parameter Tu16 iLanguages is the number of languages which are supported. iLanguages corresponds to the number of entries in the pLanguages array of LOCLANG structures.

The parameter LOCLANG *pLanguages is a pointer to an array of LOCLANG structures with iLanguages entries. For each language, the device is required to specify a set of code ranges (iRanges and iCodeRange fields of LOCLANG) which specifies which Unicode characters the strings for that language covers.

Devices are required to provide English as the first choice in the list of languages offered. For English, there has to be one code range which covers symbols 0000-007F(ASCII). ie (cLangauge="en", iRanges=1, iCodeRange[0].iStart=0, iCodeRange[0].iEnd=0x7F).

The languages will only be effective for status messages that are set after this function is called.

The following typdefs are used:

```
typdef struct {
Tu16 iStart, iEnd;
} LOCRANGE;
typdef struct {
char cLanguage[AM_LANGUAGE_LENGTH];
int iRanges;
LOCRANG iCodeRange[AM_CODERANGE_MAX];
} LOCLANG;
```

The parameter char cLanguage[AM_LANGUAGE_LENGTH] is either a two character language specification (en, da, fr, etc), or a two character language followed by a dash and a country code (en-gb, fr-ca).

The parameters int iRanges and iCodeRange[AM_CODERANGE_MAX] specify which symbol sets within Unicode are used for this particular language. For example, most Western European languages are covered by the code range 0000-00FF(Latin1). Or, if Latin1 plus some characters further into the Unicode symbols are used, the code ranges might be 0000-00FF and 0800-087F, in which case iRanges would be 2.

The JetSend protocol stack copies the data that pLanguages points to, so the local status sender should free this structure after calling this routine.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h discussion in Chapter 1 for more information.

Usage:

This function will normally be called by local status senders which support status localization after the status module has been initialized.

Requirements:

The status sender module must be initiailzed and the local status sender must support status localization. This call must come before any status messages are set which expect to use these languages. Note that status senders are required to provide English as the first choice in the list of languages provided, BUT, this is only necessary if ssSetLanguages is called. All strings specified in English may not use any other Unicode characters (only 0000-007F (ASCII) are allowed).

See also:

ssInitialize
ssShutdown
ssLocDeviceRequestEvent
ssLocJobRequestEvent
ssSetDeviceStatus
ssSetJobStatus

Samples:

```
/* note: in the following function, code range values are "made
up", and could only match actual Unicode values through utter
coincidence, or divine intervention. */
LOCLANG *getEnPlFrLOCLANG()
{
    LOCLANG *newLOCLANG, *inc;
    LOCLANG tmp;

    newLOCLANG = (LOCLANG *)Tmalloc(sizeof(LOCLANG) * 3);
    inc = newLOCLANG;

    /* create english */
    strcpy(tmp.cLanguage, "en");
    tmp.iRanges = 1;
    tmp.iCodeRange[0].iStart = 0;
    tmp.iCodeRange[0].iEnd = 127;
    memcpy(inc, &tmp, sizeof(LOCLANG));
    inc++;

    /* create polish */
    strcpy(tmp.cLanguage, "pl");
    tmp.iRanges = 2;
    tmp.iCodeRange[0].iStart = 0;
    tmp.iCodeRange[0].iEnd = 127;
    tmp.iCodeRange[1].iStart = 5150;
    tmp.iCodeRange[1].iEnd = 6160;
    memcpy(inc, &tmp, sizeof(LOCLANG));
    inc++;

    /* create french */
    strcpy(tmp.cLanguage, "fr");
    tmp.iRanges = 1;
    tmp.iCodeRange[0].iStart = 0;
    tmp.iCodeRange[0].iEnd = 127;
    memcpy(inc, &tmp, sizeof(LOCLANG));

    return(newLOCLANG);

}

...

ssInitialize();
...
LOCLANG *pLocLanguages = getEnPlFrLOCLANG();
ssSetLanguages(3, pLocLanguages);
...
```

Identity: ssLocDeviceRequestEvent

Description: This event occurs on the local status sender when the remote status observer requests a status message in a language different than the in-line language set with ssSetDeviceStatus.

Detail:
```
AMEVENT ssLocDeviceRequestEvent(char *pLanguage, REQID iRequest)
```

Relevant files: am_ss.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
appl_em.h
appl_emlib.h
tops_am_job.h

Event Structure Fields: The field char *pLanguage identifies the language the remote status observer wants the message in. This will correspond to the cLanguage in one of the LOCLANG structures used in the call to ssSetLanguages.

The local status sender should call Tfree to free the memory occupied by pLanguage, or call amFreeEvent to free the entire event structure and data.

The field REQID iRequest is used in subsequent calls to ssSendLanguage.

Usage: This event occurs on the local localized status sender to indicate a localized status observer has requested the device status string in another language. After this event, the local status sender should call ssSendLanguage to send the appropriate status string to the remote status observer.

Requirements: A local status sender must be localized and have its status sender activity initialized. Additionally, a session must be established with a remote status observer. A remote status observer may only request messages in languages which have been previously set by the local status sender via ssSetLanguages.

See also: ssInitialize
ssShutdown
ssLocJobRequestEvent
ssSetLanguages
ssSendLanguage
srSetLanguage

Samples:

```
amInitialize();
ssInitialize();
ssSetLanguages();
/* Handle amSessionStartEvent */
ssSetDeviceStatus();
/* Handle ssLocDeviceRequestEvent */
ssSendLanguage();
```

Identity: ssLocJobRequestEvent

Description: This event notifies the local status sender that a remote job status observer has requested a job status message in a language other than the in-line language provided in the ssSetJobStatus call.

Detail:
```
AMEVENT ssLocJobRequestEvent(JOBID iJob, char *pLanguage, REQID
iRequest)
```

Relevant files: am_ss.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

appl_em.h

appl_emlib.h

tops_am_job.h

Event Structure Fields: The field JOBID iJob identifies the job whose status is being requested in another language.

The field char *pLanguage identifies the language the remote status observer wants the message in. This will correspond to the cLanguage in one of the LOCLANG structures used in the call to ssSetLanguages.

The local status sender should call Tfree to free the memory occupied by pLanguage, or call amFreeEvent to free the entire event structure and its data.

The field REQID iRequest is used in subsequent calls to ssSendLanguage.

Usage: This event occurs on the local localized status sender to indicate a remote localized status observer has requested the job status string in another language. After this event, the local status sender should call ssSendLanguage, which will send the appropriate status string to the remote status observer device.

Requirements: A local status sender must be localized and have its status sender activity initialized. Additionally, a session must be established with a remote status observer. A remote status observer may only request messages in languages which have been previously set by the local status sender via ssSetLanguages

See also: ssInitialize

ssShutdown

ssLocDeviceRequestEvent

ssSetLanguages

ssSendLanguage

srSetLanguage

Samples:

```
amInitialize();
ssInitialize();
ssSetLanguages();
prInitialize();
/* Handle amSessionStartEvent */
/* HandleprJobStartEvent */
ssSetJobStatus();
/* Handle ssLocJobRequestEvent */
ssSendLanguage();
```

Identity: ssSendLanguage

Description: This call is a reply by the local status sender to either an ssLocDeviceRequestEvent or ssLocJobRequestEvent. It sends the status message in the requested language to a remote status observer.

Detail:

```
DEVAPI Tstatus ssSendLanguage(REQID iRequest, LOCTEXT *pStatus)
```

Relevant files: am_ss.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

appl_em.h

appl_emlib.h

tops_am_job.h

Parameters:
Return values: The parameter REQID iRequest corresponds to the request id returned in the ssLocDeviceRequestEvent or ssLocJobRequestEvent to which this is a response.

The parameter LOCTEXT *pStatus is a LOCTEXT structure which identifies the language and contains a Unicode string with the status message.

The following definition is used:

```
typdef struct {
char cLanguage[AM_LANGUAGE_LENGTH];
Tu16 *pString;
} LOCTEXT;
```

The variable cLanguage will match the pLanguage value returned in the corresponding ssLocJobRequestEvent or ssLocDeviceRequestEvent, and pString should be a Unicode status message in the specified language.

The JetSend protocol stack will copy the data the two pointesr point to, so the local status sender should free this memory after making a call to ssSendLanguage.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h discussion in Chapter 1 for more information.

Usage: This function is a reply by the local status sender to an earlier ssLocJobRequestEvent or ssLocDeviceRequestEvent. It will result in a corresponding srJobStatusEvent or srDeviceStatusEvent on the remote tatus observer which requested the status message in the specified language.

Requirements: This function should only be called by the local status sender in response to a previous ssLocJobRequesetEvent or ssLocDeviceRequestEvent.

See also:

srInitialize
srShutdown
ssSetLanguages
ssLocDeviceRequestEvent
ssLocJobRequestEvent
srDeviceStatusEvent
srJobStatusEvent

Samples:

```
amInitialize();
ssInitialize();
ssSetLanguages();
prInitialize();
/* Handle amSessionStartEvent */
/* Handle prJobStartEvent */
...
ssSetJobStatus();
/* Handle ssLocJobRequestEvent */
ssSendLanguage();
```

Identity: ssSetDeviceStatus

Description: This routine can be used by the local status sender to set, update, or delete a component of the device's overall status.

Detail:

```
DEVAPI Tstatus ssSetDeviceStatus(Tu16 iComponentId, void
*pComponent)
```

Relevant files: am_ss.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

appl_em.h

appl_emlib.h

tops_am_job.h

Parameters :
Return values:

The parameter Tu16 iComponentId specifies which component is being set. It can be one of the following values: STAT_DEV_VALUE or STAT_DEV_STRING.

The parameter void *pComponent is the actual value or string component.

When iComponentId is STAT_DEV_VALUE, pComponent is not a pointer, but rather the actual status value cast to (void*). It may take on four possible values:

(void*)STAT_DEV_READY - The device is ready to perform its primary function.

(void*)STAT_DEV_BUSY - The device is currently performing its primary function.

(void*)STAT_DEV_OPERATOR - The device requires an operator's assistance.

(void*)STAT_DEV_BROKEN - The device is not able to perform its primary function.

When iComponentId is STAT_DEV_STRING, pComponent is a (LOCTEXT) pointer to the textual status of the device. If the local status sender is localized, the LOCTEXT structure will specifiy the language and contain the Unicode string (16-bit) with the textual status.

The following definition is used:

```
typdef struct {
charcLanguage[AM_LANGUAGE_LENGTH];
Tu16 *pString;
} LOCTEXT;
```

If the local status sender is not localized, the LOCTEXT structure will simply be a character string (8-bit). See Localization for more information.

The JetSend protocol stack copies the information pComponent points to, so the local status sender should free this memory after calling ssSetDeviceStatus.

If pComponent is NULL, the specified device status component will be deleted.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h discussion in Chapter 1 for more information.

Usage: This routine can be used by the local status sender to set, update, or delete a component of the device's overall status. When a session is established with a remote status observer with an initialized status receiver activity, changes made to the local status surface via ssSetDeviceStatus will produce an srDeviceStatusEvent on the remote status observer. If ssSetDeviceStatus is called before a session is established, the remote status observer will receive the corresponding srDeviceStatusEvent after the session starts. The remote status observer will specify which components it is interested in when it calls srInitialize. Device status updates made by the local status sender will automatically be sent the the remote status observer until the observer is no longer interested in receiving status information and calls srReleaseDeviceStatus. The local status sender receives no notification that a remote status observer is no longer interested in status information.

Requirements: This function may only be called by a local status sender after a successful ssInitialize and before an ssShutdown. It may be called before or after a session is established with a remote status observer.

See also: ssInitialize

ssShutdown

srDeviceStatusEvent

Localization

Samples:

```
amInitialize();
ssInitialize();
ssSetLanguages()
ssSetDeviceStatus(STAT_DEV_VALUE, (void*)STAT_DEV_READY);
amStartSession();
...
ssSetDeviceStatus(STAT_DEV_VALUE, (void*)STAT_DEV_BROKEN);
```

Identity: ssSetJobStatus

Description: This routine is used by the local status sender to set, update, or delte a component of the job status.

Detail:
```
DEVAPI Tstatus ssSetJobStatus(JOBID iJob, Tu16 iComponentId, void
*pComponent)
```

Relevant files: am_ss.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

appl_em.h

appl_emlib.h

tops_am_job.h

Parameters:
Return values:

The parameter JOBID iJob is the identifer for the push job whose status is being set. This corresponds to the job id returned in the prJobStartEvent.

The parameter Tu16 iComponentId specifies which component is being set. It can be one of the following values: STAT_JOB_VALUE or STAT_JOB_STRING.

The parameter void *pComponent is the actual value or string component.

When iComponentId is STAT_JOB_VALUE, pComponent is not a pointer, but rather the actual status value cast to (void*). It may take on five possible values:

(void*)STAT_JOB_OK - The job has been successfully processed.

(void*)STAT_JOB_FAIL - The job has been cancelled or something failed during processing.

(void*)STAT_JOB_PENDING - The job has been put on hold and will be processed later.

(void*)STAT_JOB_PROGRESS - The job is being processed.

(void*)STAT_JOB_PERCENTAGE - The job is being processed, plus a value between 0 and 100 has been added to STAT_JOB_PERCENTAGE to indicate how far the job has progressed. For example, (void*)(STAT_JOB_PERCENTAGE + 30) would indicate the job is 30 percent done.

When iComponentId is STAT_JOB_STRING, pComponent is a (LOCTEXT) pointer to the textual status of the job. If the local status sender is localized, the LOCTEXT structure will specifiy the language and contain the Unicode string (16-bit) with the textual status.

The following definition is used:

```
typdef struct {
charcLanguage[AM_LANGUAGE_LENGTH];
Tu16 *pString;
} LOCTEXT;
```

If the local status sender is not localized, the LOCTEXT structure will simply be a character string (8-bit). See Localization for more information.

The JetSend protocol stack copies the information pComponent points to, so the local status sender should free this memory after calling ssSetJobStatus.

If pComponent is NULL, the specified device status component will be deleted. It is recommended that the local status sender does not delete the status for a job that has not yet ended (via prEndJob).

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h discussion in Chapter 1 for more information.

Usage:

This routine may be used by the local status sender to set, update or delete a component of the job status. A local status sender will use ssSetJobStatus in conjunction with the push receive activity to inform the remote status observer of the progress of a particular job. The local status sender calling ssSetJobStatus will produce an srJobStatusEvent on the remote status observer (the push sender/status observer). The remote status observer will specify which components it is interested in when it calls srInitialize. When the remote status observer is no longer intersted in the status for a given job, it will call srReleaseJobStatus. This will produce an ssJobStatusReleasedEvent on the local status sender, indicating that it can call ssSetJobStatus to to delete the released component. The local status sender can delete any component at any time, but it is recommended that a given compenent is deleted only after it has been released or the job has ended.

Requirements:

This function may only be called by a local status sender after a successful ssInitialize and before an ssShutdown. A job status can only be created by a local status sender which is also a local push receiver, and only while a job is in progress. JOBID iJob is the indentifer for the job.

See also:

ssInitialize

ssShutdown

srJobStatusEvent

srReleaseJobStatus

ssJobStatusReleasedEvent

prJobStartEvent

Samples:

```
amInitialize();
ssInitialize();
ssSetLanguages();
prInitialize();
/* Handle amSessionStartEvent */
/* Handle prJobStartEvent */
prRequestContent();
 /* Handle prContentEvent */
...
ssSetJobStatus(STAT_JOB_VALUE, (void*)(STAT_JOB_PROGRESS));
...
ssSetJobStatus(STAT_JOB_VALUE, (void*)(STAT_JOB_OK));
```

Identity: ssJobStatusReleasedEvent

Description: This event tells the local status sender that the status for a job has been released by the remote status observer, and the status component may now be deleted.

Detail:
```
AMEVENT ssJobStatusReleasedEvent(JOBID iJob, Tu16 iComponentId)
```

Relevant files: am_ss.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
appl_em.h
appl_emlib.h
tops_am_job.h

Event Structure Fields: The field JOBID iJob is the identifer for the push job whose status is being released. This corresponds to the job id used in the ssSetJobStatus call.

The field Tu16 iComponentId indicates which component is being released. This will be either STAT_JOB_VALUE or STAT_JOB_STRING.

Usage: This event tells the local status sender that the status for a job has been released by the remote status observer and can be deleted using ssSetJobStatus. The local status sender can maintain a history of old job statuses by not deleting the job status when it is released -- other remote status observers will be able to view the status sheet in its entirety which will contain the old jobs. However, this will consume resources by maintaining information which is normally freed when the job status component is deleted by the local status sender.

Requirements: This event occurs on a local status sender in response to an srReleaseJobStatus called on a remote status observer. It will only occur for a job status component which has been set by the local status sender with ssSetJobStatus.

See also: ssInitialize
ssShutdown
ssSetJobStatus
srJobStatusEvent
srReleaseJobStatus

Samples:

```
amInitialize();
ssInitialize();
prInitialize();
/* Handle amSessionStartEvent */
/* Handle prJobStartEvent */
prRequestContent();
/* Handle prContentEvent */
...
ssSetJobStatus(STAT_JOB_VALUE, (void*)(STAT_JOB_OK));
...
/* Handle ssJobStatusReleaseEvent */
ssSetJobStatus(STAT_JOB_VALUE, NULL);
```

Identity: srInitialize

Description: This routine is used by the local status observer to initialize the status receiver activity.

Detail:

```
DEVAPI Tstatus srInitialize(Tu16 iFetchMask)
```

Relevant files: am_sr.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

appl_em.h

appl_emlib.h

tops_am_job.h

Parameters:
Return values: The parameter Tu16 iFetchMask indicates what types of status should be fetched from remote status senders. The mask is constructed by ORing a combination of these values:

STAT_DEV_VALUE - fetch the machine interpretable device status of remote status senders.

STAT_DEV_STRING - fetch the device status string of remote status senders.

STAT_JOB_VALUE - fetch the machine interpretable job status of remote status senders.

STAT_JOB_STRING - fetch the job status string of remote status senders.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h discussion in Chapter 1 for more information.

Usage: Typically, devices which implement the push send activty may implement the status observer because it is the way to determine whether a sent job was successfully processed or not. Initializing the status receiver activity will allow a status observing device to request status information from remote status senders whenever a session is established.

Requirements: This routine may only be called by a local status observer after a successful amInitialize and before an amShutdown. After srInitialize returns successfully, it should not be called again unless there has been an srShutdown.

See also: amInitialize

amShutdown

srShutdown

Samples: If the local status observer wishes to fetch all available status information from remote status senders, it will call:

```
srInitialize(STAT_DEV_VALUE | STAT_DEV_STRING | STAT_JOB_VALUE |
STAT_JOB_STRING);
```

If the local status observer only wishes to receive device status information, it will call:

```
srInitialize(STAT_DEV_VALUE | STAT_DEV_STRING);
```

Identity: srShutdown

Description: This routine is used by the local status observer to shut down the status receiver activity and free any associated resources.

Detail:
```
DEVAPI Tstatus srShutdown(void)
```

Relevant files: am_sr.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
appl_em.h
appl_emlib.h
tops_am_job.h

Parameters : This routine takes no arguments.

Return values: This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h discussion in Chapter 1 for more information.

Usage: This routine is used by the local status observer to shut down the status receiver activity and free any associated resources.

Requirements: This routine may only be called by a local status observer after a successful amInitialize and before an amShutdown. The function srInitialize must also return successfully before this routine may be called.

See also: amInitialize
amShutdown
conInitialize

Samples:
```
amInitialize();
srInitialize();
...
srShutdown();
amShutdown();
```

Identity: srSetLanguage

Description: This function is used by a ocalized status observers to set the preferred language for status strings.

Detail:
```
DEVAPI Tstatus srSetLanguage(LOCLANG *pLanguage)
```

Relevant files: am_sr.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

appl_em.h

appl_emlib.h

tops_am_job.h

Parameters:
Return values: The parameter LOCLANG *pLanguage is a pointer to a LOCLANG structure. The local status observer is required to specify a set of code ranges (iRanges and iCodeRange fields of LOCLANG) which specifies which Unicode characters the strings for the language covers.

The following typdefs are used:

```
typdef struct {
Tu16 iStart, iEnd;
} LOCRANGE;


typdef struct {
char cLanguage[AM_LANGUAGE_LENGTH];
int iRanges;
LOCRANG iCodeRange[AM_CODERANGE_MAX];
} LOCLANG;
```

The variable char cLanguage[AM_LANGUAGE_LENGTH] is either a two character language specification (en, da, fr, etc), or a two character language followed by a dash and a country code (en-gp, fr-ca).

The variables int iRanges and iCodeRange[AM_CODERANGE_MAX] specify which symbol sets within Unicode are used for this particular language. For example, most Western European languages are covered by the code range 0000-00FF(Latin1). Or, if Latin1 plust some characters further into the Unicode symbols are used, the code ranges might be 0000-00FF and 0800-087F, in which case iRanges would be 2.

The JetSend protocol stack copies the data that pLanguage points to, so this memory should be freed by the local status observer after the call to srSetLanguage.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h discussion in Chapter 1 for more information.

Usage: This function will be used by localized status observers to set the preferred language for status messages. If the preferred language is not available from a given remote status sender, the local status observer will receive messages in either English ("en") or unspecified (cLanguage is the empty string, indicating the remote tatus sender is not localized). A local status sender will only receive messages containing characters that fall within the code range(s) specified in the call to srSetLanguage. However, it should always be prepared to receive messages in English or unspecified (the mandatory status language).

Requirements: This function will only be used by a localized status observer (the status receiver module must be initialized). It will normally be called just after initialization of the status receiver activity.

See also:
srInitialize
srShutdown
srDeviceStatusEvent
srJobStatusEvent
ssSetLanguages
ssSendLanguage

Samples:

```
LOCLANG *getPlLOCLANG()
{
    LOCLANG *newLOCLANG;

    newLOCLANG = (LOCLANG *)Tmalloc(sizeof(LOCLANG) * 1);

    /* create polish */
    strcpy(newLOCLANG->cLanguage, "pl");
    newLOCLANG->iRanges = 2;
    newLOCLANG->iCodeRange[0].iStart = 0;
    newLOCLANG->iCodeRange[0].iEnd = 127;
    newLOCLANG->iCodeRange[1].iStart = 5150;
    newLOCLANG->iCodeRange[1].iEnd = 6160;

    return(newLOCLANG);
}

LOCLANG *pLanguage=getPlLOCLANG();

amInitialize();
srInitialize();
srSetLanguage(pLanguage);
```

Identity: srDeviceStatusEvent

Description: This event notifies the local status observer that a component of the device status of a remote status sender has arrived, changed, or been deleted.

Detail:

```
AMEVENT srDeviceStatusEvent(SHANDLE hSession, Tu16 iComponentId,
void *pComponent)
```

Relevant files: am_sr.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

appl_em.h

appl_emlib.h

tops_am_job.h

Event Structure Fields:

The field SHANDLE hSession is the session handle used to identify the remote status sender.

The field Tu16 iComponentId indicates which component has arrived. This may be either STAT_DEV_VALUE or STAT_DEV_STRING.

The field void *pComponent is the actual value or string component.

When iComponentId is STAT_DEV_VALUE, pComponent is not a pointer, but rather the actual status value cast to (void*). It may take on four possible values:

- STAT_DEV_READY - The device is ready to perform its primary function.
- STAT_DEV_BUSY - The device is currently performing its primary function.
- STAT_DEV_OPERATOR - The device requires an operator's assistance.
- STAT_DEV_BROKEN - The device is not able to perform its primary function.

When iComponentId is STAT_DEV_STRING, pComponent is a (LOCTEXT) pointer to the textual status of the remote status sender. If the local status reciever is localized, the LOCTEXT structure will specifiy the language and contain the Unicode string (16-bit) with the textual status.

The following definition is used:

```
typdef struct {
charcLanguage[AM_LANGUAGE_LENGTH];
Tu16 *pString;
} LOCTEXT;
```

If the specified language is an empty string, it signifies the remote status sender is not localized. If the local status observer is not localized, the LOCTEXT structure will simply be a character string (8-bit). See Localization for more information.

If pComponent is NULL, it indicates that the remote component has been deleted and is no longer valid.

The local status observer should free the data that pComponent points to, or call amFreeEvent to free this data and the event structure. Note that if the component is localized, the local status sender should free both the string in the LOCTEXT structure and the structure itself.

Usage:

After a session is established between a the local status observer and a remote status sender, any ssSetDeviceStatus call on the remote status sender will result in an srDeviceStatusEvent at the local status observer. The local status observer will only receive information about the device status components specified in the iFetchMask in srInitialize. A local status observer will continue to receive information about a given component until: srReleaseDeviceStatus is called; the remote status sender deletes the component; or the session is ended.

Requirements:

This event will only occur on the local status observer after a session is established with a remote status sender which has called ssSetDeviceStatus for a component specified in the local status observer's srInitialize iFetchMask.

See also:

srInitialize

srShutdown

ssSetDeviceStatus

srReleaseDeviceStatus

Samples:

```
amInitialize();
srInitialize();
/* Handle amSessionStartEvent */
/* Handle srDeviceStatusEvent */
/* Handle srDeviceStatusEvent */
```

Identity: srReleaseDeviceStatus

Description: This routine is used by the local status observer to notify a remote status sender that it is no longer interested in device status updates for a specific device status component.

Detail:
```
DEVAPI Tstatus srReleaseDeviceStatus(SHANDLE hSession, Tu16
iComponentId)
```

Relevant files: am_sr.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

appl_em.h

appl_emlib.h

tops_am_job.h

Parameters:
Return values: The parameter SHANDLE hSession is the session handle used to identify the remote status sender. This corresponds to the session handle in the srDeviceStatusEvent.

The parameter Tu16 iComponentId indicates which component is being released. This may be either STAT_DEV_VALUE or STAT_DEV_STRING.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h discussion in Chapter 1 for more information.

Usage: The local status observer can call this function to notify a remote status sender it is no longer interested in device status updates for the specified component.

Requirements: The local status observer must have a sesion established with a remote status sender. Additionaly, srReleaseDeviceStatus should only be called to release a component which has already arrived in a previous srDeviceStatusEvent.

See also: srInitialize

srShutdown

ssSetDeviceStatus

srDeviceStatusEvent

Samples:
```
amInitialize();
srInitialize();
/* Handle amSessionStartEvent */
/* Handle srDeviceStatusEvent with particular attention to
STAT_DEV_VALUE */
/* Handle srDeviceStatusEvent with particular attention to
STAT_DEV_STRING */
srReleaseDeviceStatus(..., STAT_DEV_STRING, ...);
/* Handle srDeviceStatusEvent with particular attention to
STAT_DEV_VALUE */
srReleaseDeviceStatus(..., STAT_DEV_VALUE, ...);
```

Identity: srJobStatusEvent

Description: This event notifes the local status observer that a component of the job status message has arrived.

Detail:

```
AMEVENT srJobStatusEvent(JOBID iJob, Tu16 iComponentId, void
*pComponent)
```

Relevant files: am_sr.c

tops.h

tops_device.h

tops_am.h

tops_am_list.h

appl_em.h

appl_emlib.h

tops_am_job.h

Event Structure Fields:

The field JOBID iJob is the identifer for the push job whose status is being received. This corresponds to the job id returned in the psStartJob call.

The field Tu16 iComponentId specifies which component is being set. It can be one of the following values: STAT_JOB_VALUE or STAT_JOB_STRING.

The field void *pComponent is the actual value or string component.

When iComponentId is STAT_JOB_VALUE, pComponent is not a pointer, but rather the actual status value cast to (void*). It may take on five possible values:

(int)STAT_JOB_OK - The job has been successfully processed.

(int)STAT_JOB_FAIL - The job has been cancelled or something failed during processing.

(int)STAT_JOB_PENDING - The job has been put on hold and will be processed later.

(int)STAT_JOB_PROGRESS - The job is being processed.

(int)STAT_JOB_PERCENTAGE - The job is being processed, plus a value between 0 and 100 has been added to STAT_JOB_PERCENTAGE to indicate how far the job has progressed. For example, (int)(STAT_JOB_PERCENTAGE + 30) would indicate the job is 30 percent done.

When iComponentId is STAT_JOB_STRING, pComponent is a (LOCTEXT) pointer to the textual status of the job. If the local status observer and remote status sender are localized, the LOCTEXT structure will specifiy the language and contain the Unicode string (16-bit) with the textual status.

The following definition is used:

```
typdef struct {
charcLanguage[AM_LANGUAGE_LENGTH];
Tu16 *pString;
} LOCTEXT;
```

If the specified language is an empty string, it signifies the remote status sender is not localized. If the local status receiver is not localized, the LOCTEXT structure will simply be a character string (8-bit). See Localization for more information.

The JetSend protocol stack copies the information pComponent points to, so the local status observer should free this memory, or call amFreeEvent to free the memory along with the event structure.

If pComponent is NULL, it indicates that the remote component has been deleted and is no longer valid.

The local status observer should free the data that pComponent points to. Note that if the component is localized, the device should free both the string in the LOCTEXT structure and the structure itself.

Usage:

This event notifies a local status observer of the status of a push job. After a job starts, the remote status sender may call ssSetJobStatus. This will result in an srJobStatusEvent on the local status observer, if the set component is one specified in the srInitialize iFetchMask. This event can arrive for a given job even after that job has ended. This feature can be used by the remote status sender/push receiver to notify the local status observer/push sender of the final job status. The local status observer can call srRelaseJobStatus to notify the remote status sender it is no longer interested in a specific job status component for a given job.

Requirements:

An srJobStatusEvent can only occur after on the local status sender after a job has been started. The local status observer must have its push sender activity initialized, while the remote status sender must have its push receiver activity initialized. The remote status sender must call ssSetJobStatus; this will produce an srJobStatusEvent on the local status observer.

See also:

srInitialize

srShutdown

psStartJob

ssSetJobStatus

srReleaseJobStatus

Samples:

```
amInitialize();
psInitialize();
srInitialize();
/* Handle amSessionStartEvent */
/* Handle srDeviceStatusEvent with particular attention to
STAT_DEV_VALUE */
/* Handle srDeviceStatusEvent with particular attention to
STAT_DEV_STRING */
psStartJob();
/* Handle srJobStatusEvent */
/* Handle psJobEndEvent */
/* Handle srJobStatusEvent */
srReleaseJobStatus();
```

Identity: srReleaseJobStatus

Description: This routine lets the local status observer reject a job status surface if it is not interested in further events regarding the specified job component for the specified job.

Detail:
```
DEVAPI Tstatus srReleaseJobStatus(JOBID iJob, Tu16 iComponentId)
```

Relevant files: am_sr.c
tops.h
tops_device.h
tops_am.h
tops_am_list.h
appl_em.h
appl_emlib.h
tops_am_job.h

Parameters: The parameter JOBID iJob is the identifer for the push job whose status is being released. This corresponds to the job id in the srJobStatusEvent and the psStartJob call.

The parameter Tu16 iComponentId indicates which component is being released. This will be either STAT_JOB_VALUE or STAT_JOB_STRING.

Return values: This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h discussion in Chapter 1 for more information.

Usage: A local status observer can reject a job status surface if it is not interested in further events regarding the specified job component for the specified job. This does not mean that the local status observer is cancelling the job, just that the status observer is no longer interested in a particular status component.

Requirements: The local status observer must have a sesion established with a remote status sender. Additionaly, srReleaseJobStatus should only be called to release a component which has already arrived in an srJobStatusEvent.

See also: srInitialize
srShutdown
srJobStatusEvent
ssSetJobStatus

Samples:

```
amInitialize();
psInitialize();
srInitialize();
/* Handle mSessionStartEvent */
/* Handle srDeviceStatusEvent with particular attention to
STAT_DEV_VALUE */
/* Handle srDeviceStatusEvent with particular attention to
STAT_DEV_STRING */
psStartJob();
/* Handle srJobStatusEvent */
srReleaseJobStatus();
```

Appendix G

Address Send/Address Receive Activity

Part of Device Independent API

Identity: asInitialize

Description: This routine initializes the sender side of the address activity.

Detail:
```
DEVAPI Tstatus asInitialize(void)
```

Relevant files: am_as.c
tops.h
tops_device.h
tops_am.h
appl_queue.h
appl_em.h
tops_oiplib.h

Parameters : This routine takes no arguments.

Return values:

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h discussion in Chapter 1 for more information.

Usage: Any local device wishing to use the sending side of the address activity will call this routine. It will often be called immediately following amInitialize. It must return successfully before any other address sender activity routines may be called.

Requirements: This routine may only be called after a successful amInitialize and before an amShutdown. After asInitialize returns successfully, it should not be called again unless there has been an asShutdown.

See also: amInitialize
amShutdown
asShutdown

Samples:
```
amInitialize();
asInitialize();
```

Identity: asShutdown

Description: This routine shuts down the sending side of the address activity and frees any associated resources.

Detail:
```
DEVAPI Tstatus asShutdown(void)
```

Relevant files: am_as.c
tops.h
tops_device.h
tops_am.h
appl_queue.h
appl_em.h
tops_oiplib.h

Parameters: This routine takes no arguments.

Return values:

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h discussion in Chapter 1 for more information.

Usage: This routine will be called when a local address sending device wants to shut down the address sender activity and free its resources.

Requirements: This routine may only be called after a successful amInitialize and before an amShutdown. asInitialize must also return successfully before this routine may be called.

See also: amInitialize
amShutdown
asInitialize

Samples:
```
amInitialize();
asInitialize();
...
asShutdown();
amShutdown();
```

Identity: asSurfaceArrivedEvent

Description: This event notifies the local address sender that the address surface of a remote address receiver has arrived, meaning the local address sender may now send address cards to the remote address receiver.

Detail:
```
AMEVENT asSurfaceArrivedEvent(SHANDLE hSession)
```

Relevant files: am_as.c
tops.h
tops_device.h
tops_am.h
appl_queue.h
appl_em.h
tops_oiplib.h

Event Structure Fields: The field SHANDLE hSession is the session handle used to identify the remote address receiver.

Usage: After a session has been established, the local address sender will receive an asSurfaceArrivedEvent after it initializes its address sender activity and the remote address receiver initializes its address receiver activity. The local address sender will receive no notification if the remote device has no address receiver activity initialized. This event will not occur after an asShutdown on the local address sender or an arShutdown on the remote address receiver. The event will be posted to the AM event queue and should be retrieved with amGetNextEvent. No address send routines can send information to a remote address receiver until its session handle arrives in an asSurfaceArrivedEvent. A local address sender can continue to use a remote receiver's address surface until an asSurfaceGoneEvent arrives for the remote address receiver.

Requirements: A session must be established between a local address sender and a remote address receiver who each have the respective activity initialized.

See also: amStartSession
amSessionStartEvent
asInitialize
asShutdown
arInitialize
arShutdown
asSendCard
asCardSendEvent
asSendSelf

Samples:

```
amInitialize();
conInitialize();
asInitialize();
amStartSession();
/* Handle amSessionStartEvent */
/* Handle asSurfaceArrivedEvent */
asSendSelf();
asSendCard();
/* Handle asCardSentEvent */
/* Handle asSurfaceGoneEvent */
```

Identity: asSurfaceGoneEvent

Description: This event notifies the local address sender that the address surface of a remote address receiver has been unimpressed, meaning the local address sender can no longer send address cards to the remote address receiver.

Detail:
```
AMEVENT asSurfaceGoneEvent(SHANDLE hSession)
```

Relevant files: am_as.c
tops.h
tops_device.h
tops_am.h
appl_queue.h
appl_em.h
tops_oiplib.h

Event Structure Fields: The field SHANDLE hSession is the session handle used to identify the remote address receiver. This corresponds to the session handle returned in a previous asSurfaceArrivedEvent.

Usage: This event notifies the local address sender that the address surface of the remote address receiver identified by hSession has been removed. This occurs when the remote address receiver shuts down its address receive activity or the session ends.

Requirements: An asSurfaceGoneEvent will only occur on the local address sender after receiving an asSurfaceArrivedEvent and before the local address sender shuts down its address send activity via asShutdown or amShutdown.

See also: asInitialize
asShutdown
asSurfaceArrivedEvent

Samples:
```
amInitialize();
amStartSession();
/* Handle amSessionStartEvent */
/* Handle asSurfaceArrivedEvent */
asSendCard()
/* Handle asCardSentEvent */
/* Handle asSurfaceGoneEvent */
```

Identity: asSendCard

Description: This routine is used by the local address sender to send an address card to a remote address receiver.

Detail:
```
DEVAPI Tstatus asSendCard(ADDRESS *pData, SHANDLE hTarget)
```

Relevant files: am_as.c

tops.h

tops_device.h

tops_am.h

appl_queue.h

appl_em.h

tops_oiplib.h

Parameters:
Return values: The parameter ADDRESS *pData points to a structure cointaing the address information to be sent. The structure is defined in tops_device.h and consists of:

typdef struct {

char* pAddress: ....; /* a..../* (char*) pointing to the textual string containing the address. */

char* pLabel: ....; /* a..../* (char*) pointing to the textual label of the card. */

BD* emIcon: ....; /* a..../* (BD*) which is the e-material icon. */

BD* emEncodings: ....; /* a..../* (BD*) which is a set of e-material encodings. */

} ADDRESS;

Some of the information can be left out by setting the field to NULL, but the pAddress field should always be available.

The JetSend protocol stack will free the memory associated with pData as the card is sent.

The parameter SHANDLE hTarget is the session handle of the remote address receiver. This corresponds to a session handle returned in an asSurfaceArrivedEvent.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h discussion in Chapter 1 for more information.

Usage: This function allows the local address sender to send an address card to a remote address receiver.

Requirements: Before sending a card to a remote address receiver, the local address sender must first have received an asSurfaceArrivedEvent corresponding to the target receiver. A card may not be sent to a remote address receiver after an asSurfaceGoneEvent arrives from that device. Additionally, only one card may be sent at a time, ie after calling asSendCard, the local address sender must wait for the corresponding asCardSentEvent before again calling asSendCard.

See also:

asInitialize

asShutdown

asSurfaceArrivedEvent

asSurfaceGoneEvent

asCardSentEvent

arCardReceivedEvent

Samples:

```
amInitialize();
conInitialize();
asInitialize();
amStartSession(ip:XXX.XXX.XXX.010);
/* Handle amSessionStartEvent with session handle of 2 */
/* Handle asSurfaceArrivedEvent with session handle of 2 */
amStartSession(ip:XXX.XXX.XXX.030)
/* Handle amSessionStartEvent with session handle of 3 */
/* Handle asSurfaceArrivedEvent with session handle of 3 */
asSendSelf(..., 2);
asSendSelf(..., 3);
asSendCard(..., 2);
/* Handle asCardSentEvent with session handle of 2 */
asSendCard(..., 3);
/* Handle asCardSentEvent with session handle of 3 */
```

Identity: asCardSentEvent

Description: This event notifies the local address sender that a card has been sent and the JetSend protocol stack is ready to send another card.

Detail:
```
AMEVENT asCardSentEvent(char *pAddress, SHANDLE hTarget, Tu16
iStatus)
```

Relevant files: am_as.c

tops.h

tops_device.h

tops_am.h

appl_queue.h

appl_em.h

tops_oiplib.h

Event Structure Fields: The field char *pAddress is the pointer to the textual string containing the address which was sent. This corresponds to the pAddress field of the pData structure in the asSendCard call.

The local address sender should free the memory which pAddress points to, or call amFreeEvent to free the entire event structure and data.

The field SHANDLE hTarget identifies which remote address receiver the address was sent to. This corresponds to the hTarget parameter in the asSendCard call.

The field Tu16 iStatus indicates which fields of the address were sent. It is constructed by ORing the component ids of the send fields, similar to the iFetchMask in conInitialize and arInitialize. If all fields are sent, iStatus will be (SELF_ADDRESS | SELF_LABEL | SELF_ICON | SELF_ENCODINGS). If none are sent, iStatus will be 0.

Usage: The local address sender will receive this event in response to a prior asSendCard call. This event notifies the device that the infrastructure is ready to send another card.

Requirements: This eventwill only occur in response to an asSendCardCall.

See also: asInitialize

asShutdown

asSurfaceArrivedEvent

conInitialize

arInitialize

Samples:

```
amInitialize();
conInitialize();
asInitialize();
amStartSession(ip:XXX.XXX.XXX.010);
/* Handle amSessionStartEvent with session handle of 2 */
/* Handle asSurfaceArrivedEvent with session handle of 2 */
amStartSession(ip:XXX.XXX.XXX.030)
/* Handle amSessionStartEvent with session handle of 3 */
/* Handle sSurfaceArrivedEvent with session handle of 3 */
asSendSelf(..., 2)
asSendSelf(..., 3)
asSendCard(..., 2)
/* Handle asCardSentEvent with session handle of 2 */
asSendCard(..., 3)
/* Handle asCardSentEvent with session handle of 3 */
```

Identity: asSendSelf

Description: This function pushes the self surface of the local address sender as an address card on a remote address receiver.

Detail:
```
DEVAPI Tstatus asSendSelf(SHANDLE hSource, SHANDLE hTarget)
```

Relevant files: am_as.c

tops.h

tops_device.h

tops_am.h

appl_queue.h

appl_em.h

tops_oiplib.h

Parameters:
Return values: The parameter SHANDLE hSource specifies the session whose *self surface* will be sent. This allows the local address sender to send the *self surface* of one remote appliance to another remote address receiver, or to send its own *self surface* to a remote address receiver as an address card. Currently, only the latter is supported, and this parameter remains unused – the local address sender's own self surface will always be sent.

The parameter SHANDLE hTarget specifies the remote address receiver to which the address card will be sent.

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h discussion in Chapter 1 for more information.

Usage: This function can be used to send a local address sender's self surface to a remote address receiver as an address card. This call will generate an arCardReceivedEvent on the remote address receiver. This call will not result in an asCardSentEvent (only the asSendCard call generates the asCardSentEvent).

Requirements: Before sending a card to a remote address receiver, the local address sender must first have received an asSurfaceArrivedEvent corresponding to the target receiver. A card may not be sent to a remote address receiver after an asSurfaceGoneEvent arrives from that device. Additionally, for the local address sender to send its own *self surface*, it must have an initialized contact module; else there is no *self surface* to send.

See also: asInitialize

asShutdown

asSurfaceArrivedEvent

asSurfaceGoneEvent

conInitialize

conShutdown

arCardReceivedEvent

Samples:

```
amInitialize();
conInitialize();
asInitialize();
amStartSession(ip:XXX.XXX.XXX.010);
/* Handle amSessionStartEvent with session handle of 2 */
/* Handle asSurfaceArrivedEvent with session handle of 2 */
amStartSession(ip:XXX.XXX.XXX.030);
/* Handle amSessionStartEvent with session handle of 3 */
/* Handle asSurfaceArrivedEvent with session handle of 3 */
asSendSelf(..., 2);
asSendSelf(..., 3);
```

Identity: arInitialize

Description: This routine is used by a local address receiver to initialize the receiving side of the address activity.

Detail:
```
DEVAPI Tstatus arInitialize(Tu16 iFetchMask)
```

Relevant files: am_ar.c
tops.h
tops_device.h
tops_am.h
appl_queue.h
appl_em.h
tops_oiplib.h

Parameters :
Return values: Th parameter Tu16 iFetchMask specifies the address components that will be requested automatically by the JetSend protocol stack when the the top level surface of an incoming address card is received. The mask is constructed by ORing component ids. Legal component ids are: SELF_ADDRESS, SELF_LABEL, SELF_ICON, SELF_ENCODINGS. For example, if the local address receiver wants only to retrieve the address and the name of the address cards it receives, it would call arInitialize(SELF_ADDRESS | SELF_LABEL).

This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h discussion in Chapter 1 for more information.

Usage: Any device wishing to use the address receiver activity will call this routine. It will often be called immediately following amInitialize. It must return successfully before any other address receiver activity routines may be called, or before any address receiver events may occur.

Requirements: This routine may only be called by the local address receiver after a successful amInitialize and before an amShutdown. After arInitialize returns successfully, it should not be called again unless there has been an arShutdown.

See also: amInitialize
amShutdown
arShutdown

Samples:
```
amInitialize();
arInitialzie();
```

Identity: arShutdown

Description: This routines is called by the local address receiver to shutdown the address receiver activity and free any associated resources.

Detail:
```
DEVAPI Tstatus arShutdown(void)
```

Relevant files: am_ar.c
tops.h
tops_device.h
tops_am.h
appl_queue.h
appl_em.h
tops_oiplib.h

Parameters :
Return values: This routine returns 0 on success. On failure, the return value may be parsed to determine the location and cause of error. See tops.h discussion in Chapter 1 for more information.

Usage: This routine will be called when a device wants to shut down the address receiver activity and free its resources.

Requirements: This routine may only be called by the local address receiver after a successful amInitialize and before an amShutdown. arInitialize must also return successfully before this routine may be called.

See also: amInitialize
amShtudown
arInitialize

Samples:
```
amInitialize();
arInitialize();
...
arShutdown();
amShutdown();
```

Identity: arSurfaceRejectedEvent

Description: This event notifies the local address receiver that its address surface has been unimpressed by a remote address sender or that the session has ended. This means the local address sender won't receive any more address cards from the remote address sender.

Detail:
```
AMEVENT arSurfaceRejectedEvent(SHANDLE hSession)
```

Relevant files: am_ar.c
tops.h
tops_device.h
tops_am.h
appl_queue.h
appl_em.h
tops_oiplib.h

Event Structure Fields: The field SHANDLE hSession indentifies the remote address sender which has unmipressed the local ADDRESS surface. This value corresponds to a session handle returned in an amSessionStartEvent.

Usage: This event notifies the local address receiver that it will not receive any address information from the remote address sender. This event signifies the remote address sender does not have an address send module installed, or the session has ended.

Requirements: First, a session must be established with a remote address sender. Second, the remote address sender must either shut down its address send activity, or the session must end. This event will not occur after an arShutdown on the local address receiver.

See also: arInitialize
arShutdown
amSessionStartEvent
amSessionEndEvent

Samples:
```
amInitialize();
arInitialize();
/* Handle amSessionStartEvent */
/* Handle arCardReceivedEvent */
/* Handle amSessionEndEvent */
/* Handle arSurfaceRejectedEvent */
```

Identity: arCardReceivedEvent

Description: This event tells the local address receiver that an address card has been received from a remote address sender.

Detail:
```
AMEVENT arCardReceivedEvent(ADDRESS *pData, SHANDLE hSender)
```

Relevant files: am_ar.c
tops.h
tops_device.h
tops_am.h
appl_queue.h
appl_em.h
tops_oiplib.h

Event Structure Fields: The field ADDRESS *pData points to a structure with the address card information. Some of the fields of that structure may be NULL, indicating they were not requested or were unavailable, but the pAddress field will always be set. The structure is of the form:

typdef struct {
char* pAddress;a (char*) pointing to the textual string containing the address.
char* pLabel;a (char*) pointing to the textual label of the card.
BD* emIcon;a (BD*) which is the e-material icon.
BD* emEncodings;a (BD*) which is a set of e-material encodings.
} ADDRESS;

The local address receiver is responsible for freeing all the memory pointed to by pData and its members, or calling amFreeEvent to free the entire event structure, including the address card and its data.

The field SHANDLE hSession indentifies the remote address sender.

Usage: This event passes address card information sent by a remote address sender to the local address receiver.

Requirements: A session must be established with a remote address sender. Additionally, this event will only occur in response to an asSendCard or asSendSelf call by the remote address sender.

See also: arInitialize
arShutdown
asSendCard
asSendSelf
arSurfaceRejectedEvent

Samples:

```
amInitialize();
arInitialize();
/* Handle amSessionStartEvent */
/* Handle arCardReceivedEvent */
...
/* Handle amSessionEndEvent */
/* Handle arSurfaceRejectedEvent */
```

Appendix H

Buffer Function Reference

Part of System API

Identity: CreateBuf

Description: Attempts to create and initialize a task block control buffer (Tbcb).

Detail:

```
Tbcb* CreateBuf( Tu32 prmSize );
```

Relevant files: tops.h

appl_buf.h

Parameters: The parameter prmSize is the size of the data space to create.

Return values:

The return value Tbcb is a pointer to a Tbcb containing prmSize bytes of data space, or NULL if memory could not be allocated.

Usage: This is used throughout core code to create a buffer that will be passed between the TOPS layers. Once the buffer is passed to another layer in the core code, it is the responsibility of the receiving layer to either free the buffer or pass it to another layer.

Note: All buffers created are aligned on a 4-byte boundary for portability. If the prmSize is not a multiple of 4, then up to 3 bytes of padding are added to the buffer to ensure this.

Requirements: The only way to correctly free the Tbcb is by calling DestroyBuf(). Otherwise, it will cause memory leaks.

See also: MustCreateBuf

DestroyBuf

ReinitializeBuf

PrependCopyToBuf

PrependBuf

UnprependBuf

BufDataSpace

BufDataLen

BufFreeSpace

BufDataAddr

Samples: JSP wants to send TIAPI a buffer of data to send to a foreign peer for the JSP protocol.

```
Tbcb * newbuff;
Tu32 ospHeaderSize = 32;
Tu32 value = 666;

newbuff=CreateBuf( ospHeaderSize );

if ( newbuff == NULL )
{
printf("  Could not allocate a new buffer\n");
return;
}

PrependCopyToBuf( newbuff, (void*)&value, sizeof(value) );
```

Identity: MustCreateBuf

Description: Always creates and initializes a task block control buffer (Tbcb).

Detail:
```
Tbcb* MustCreateBuf( Tu32 prmSize );
```

Relevant files: See CreateBuf

Parameters: The parameter Tu32prmSize is the size of the data space to create.

Return Values: The return Tbcb is a pointer to a Tbcb containing prmSize bytes of data space.

Usage: See CreateBuf

Requirements: See CreateBuf

See also:
CreateBuf
DestroyBuf
ReinitializeBuf
PrependCopyToBuf
PrependBuf
UnprependBuf
BufDataSpace
BufDataLen
BufFreeSpace
BufDataAddr

Samples: Same as CreateBuf sample, except there is no need to check if newbuff is NULL.

Identity: DestroyBuf

Description: To free a Tbcb buffer previously created with CreateBuf().

Detail:
```
void DestroyBuf( Tbcb* prmTbcbPtr );
```

Relevant files: tops.h
appl_buf.h

Parameters: The parameter Tbcb * prmTbcbPtr is a pointer to a Tbcb.

Return values:

Usage: This function is called when a layer wishes to free the memory created when CreateBuf() was called. It should be called by the function which currently owns the Tbcb.

Requirements: Prior to calling DestroyBuf(), the buffer must have been created with CreateBuf(). After the call to DestroyBuf(), the buffer is no longer usable.

See also: CreateBuf
MustCreateBuf
ReinitializeBuf
PrependCopyToBuf
PrependBuf
UnprependBuf
BufDataSpace
BufDataLen
BufFreeSpace
BufDataAddr

Samples: The buffer from the previous example is in the TIAPI layer. Assume that the device is shutting down, and TIAPI is freeing all outstanding memory. It comes upon this buffer, and therefore, calls DestroyBuf() to free the buffer.

Identity: ReinitializeBuf

Description: This function is used to "clean out" any data in the buffer basically have an empty Tbcb for use.

Detail:
```
void ReinitializeBuf( Tbcb* prmTbcbPtr );
```

Relevant files: tops.h

appl_buf.h

Parameters: The parameter prmTbcbPtr is a pointer to a Tbcb.

Return values:

Usage: This function is used when it is desired to reuse a Tbcb without having to DestroyBuf() and CreateBuf().

Requirements: Prior to this function being called, a buffer had to have been created using CreateBuf().

See also: CreateBuf

MustCreateBuf

DestroyBuf

PrependCopyToBuf

PrependBuf

UnprependBuf

BufDataSpace

BufDataLen

BufFreeSpace

BufDataAddr

Samples: JSP passes TIAPI buffers to fill for incoming data. If TIAPI returns a buffer to JSP and JSP is going to act on the buffer, but not pass it up, JSP will call ReinitializeBuf() and then PrependBuf() on the size of the buffer, and pass it back down to TIAPI for receiving incoming data.

Identity: PrependCopyToBuf

Description: Copies data into a Tbcb.

Detail:
```
int PrependCopyToBuf( Tbcb* prmTbcbPtr, void*  prmData, Tu32
prmSize );
```

Relevant files: tops.h

appl_buf.h

Parameters: The parameter prmTbcbPtr is a pointer to a Tbcb.

Return values:

The parameter prmData is a pointer to data that should be copied into the buffer.

The parameter prmSize is the number of bytes of data to copy into the buffer.

The return int will take on the following values.

BUFOK if the data was successfully prepended onto the buffer.

BUFERR if the data could not be prepended due to lack of sufficient room in the buffer.

Usage: This function is called whenever it is desired to prepend data onto the buffer. Data is prepended from the bottom of the buffer to the top, so that each layer of the core can prepend any information needed, and on the foreign side, can remove the data in the opposite order. It's like filling up a glass and then emptying it.

Note: All data prepended to a Tbcb are aligned on a 4-byte boundary for portability. Therefore, if prmSize is not a multiple of 4, then up to 3 bytes of padding are appended onto the end of the data in the buffer to ensure this.

Requirements: Prior to this function being called, the buffer had to have been created using CreateBuf().

See also: CreateBuf

MustCreateBuf

DestroyBuf

ReinitializeBuf

PrependBuf

UnprependBuf

BufDataSpace

BufDataLen

BufFreeSpace

BufDataAddr

Samples: See CreateBuf() sample code.

Identity: PrependBuf

Description: Makes room for data to be copied into a Tbcb. It does NOT copy data to the Tbcb.

Detail:
```
int PrependBuf( Tbcb* prmTbcbPtr, Tu32 prmSize );
```

Relevant files: tops.h

appl_buf.h

Parameters : The parameter prmTbcbPtr is a pointer to a Tbcb.

The parameter prmSize is the number of bytes to make room for.

Return values:

The return value int can take the following value.

BUFOK - if the data pointer was moved the requested number of bytes.

BUFERR - if there was not sufficient room for the requested number of bytes.

Usage: This function is called by PrependCopyToBuf() to move the data pointer the requested number of bytes before copying the data to the buffer. It should be called with care since data is NOT copied to the Tbcb.

Note: All prepended sizes to a Tbcb are aligned on a 4-byte boundary for portability. Therefore, if prmSize is not a multiple of 4, then up to 3 bytes of padding are added to ensure this.

Requirements: Prior to this function, the buffer must have been created with CreateBuf().

See also: CreateBuf

MustCreateBuf

DestroyBuf

ReinitializeBuf

PrependCopyToBuf

UnprependBuf

BufDataSpace

BufDataLen

BufFreeSpace

BufDataAddr

Samples: When JSP passes a buffer to TIAPI for receiving incoming data, it calls ReinitializeBuf() to reset the buffer, and then it must move the data pointer to the top of the data space in the buffer before passing it to TIAPI, (thus calling PrependBuf(newbuff, BufFreeSpace(newbuff))). TIAPI will then just have to copy to the BufDataAddr() of the buffer to fill it with incoming data.

Identity: UnprependBuf

Description: Remove a certain number of bytes from the buffer.

Detail:

```
int UnprependBuf( Tbcb* prmTbcbPtr, Tu32 prmSize );
```

Relevant files: tops.h

appl_buf.h

Parameters: The parameter prmTbcbPtr is a pointer to a Tbcb.

Return values:

The parameter prmSize is the number of bytes to "remove" from the top of the data.

The return value int- can take on the following values.

BUFOK - if the requested number of bytes was successfully "removed" from the top of the buffer.

BUFERR - if there was not sufficient room to remove the requested number of bytes.

Usage: This function is called to move the data pointer down the data stack a number of bytes.

Note: All unprepended sizes are aligned on a 4-byte boundary for portability. Therefore, if prmSize is *not* a multiple of 4, up to 3 bytes of padding will be added to insure this.

Requirements: The buffer must have been created using CreateBuf().

See also: CreateBuf

MustCreateBuf

DestroyBuf

ReinitializeBuf

PrependCopyToBuf

PrependBuf

BufDataSpace

BufDataLen

BufFreeSpace

BufDataAddr

Samples:

As in the sample for CreateBuf(), JSP had prepended ospHeaderSize data to the Tbcb. On the foreign peer side, this buffer is received in the TIAPI layer and sent up to the JSP layer (assume it's called thisbuffer). Assuming that TIAPI has unprepended any data that it had prepended on the sending side, JSP now needs to unprepend the header it had prepended on the receiving side.

```
Tu32 ospHeaderSize = 32;

/* To see the data that JSP had prepended */
ospdata=BufDataAddr( thisbuffer );
printf(" %ld\n", ospdata );   /* This should yield - 666 */

/* Now, JSP can unprepend the data and see if it needs to send
the buffer to JIP */
if ( UnprependBuf( thisbuffer, ospHeaderSize ) == BUFOK )
{
if ( BufDataSpace( thisbuffer ) == 0 )
DestroyBuf( thisbuffer );
else
/* Pass it up to JIP */
}
else
/* Log an error */
```

Identity: BufDataSpace

Description: Returns how many bytes of data are in the buffer, *including* padding.

Detail:
```
Tu32 BufDataSpace( Tbcb* prmTbcbPtr );
```

Relevant files: tops.h
appl_buf.h

Parameters : The parameter prmTbcbPtr is a pointer to a Tbcb.

Return values:

This function returns the number of bytes of data that are in the buffer, including padding.

Usage: This is used by the lower layers of core code for sending data. Padding should be sent to ensure receiving data on a 4-byte boundary.

Requirements: The buffer must have been created using CreateBuf().

See also: CreateBuf
MustCreateBuf
DestroyBuf
ReinitializeBuf
PrependCopyToBuf
PrependBuf
UnprependBuf
BufDataLen
BufFreeSpace
BufDataAddr

Samples:

When JSP sends data to TIAPI, it needs to send the padding as well, to ensure that all data is received on a 4-byte boundary.

```
typedef {
   Tu32 val;
    Tu8  valbool;
.
.
}  ospHeaderStruct;

Tbcb * newbuff;
Tu32 ospHeaderSize = sizeof( ospHeaderStruct );
ospHeaderStruct * outdata;

/* Assume ospCreateOspHeader() allocates and assigns values for
an ospheader.  Return value is ospHeaderStruct *     */
outdata=ospCreateOspHeader();

newbuff=CreateBuf( ospHeaderSize );

PrependCopyToBuf( newbuff, (void*)outdata, ospHeaderSize );

/* Now JSP wants to send this, including any padding to a foreign
peer via TIAPI */
xptSend( …, …., newbuff, BufDataSpace( newbuff ) );
```

Identity: BufDataLen

Description: Returns how many bytes of data are in the buffer, NOT including the first padding.

Detail:
```
Tu32 BufDataLen( Tbcb* prmTbcbPtr );
```

Relevant files: tops.h
appl_buf.h

Parameters: The parameter prmTbcbPtr is a pointer to a Tbcb.

Return values: This function returns the number of bytes of data, not including the first padding that was added.

Usage: This is used by upper layers of core and the device to determine the number of bytes of data in the buffer.

Requirements: The buffer must have been created using CreateBuf().

See also: CreateBuf
MustCreateBuf
DestroyBuf
ReinitializeBuf
PrependCopyToBuf
PrependBuf
UnprependBuf
BufDataSpace
BufFreeSpace
BufDataAddr

Samples:

Assume the device on one side has data to send to another device.

AM will call CreateBuf() and PrependCopyToBuf() the data into a Tbcb. This buffer will be passed to JIP, which will PrependCopyToBuf() any information it needs to receive on the other side. JIP will then send the buffer to JSP, which will PrependCopyToBuf() it's header as well. Finally, the buffer will be sent to TIAPI which could PrependCopyToBuf() any header information prior to sending the buffer to the other device.

On the receiving side, TIAPI will copy the data into a buffer (thisbuff) that JSP has sent down to fill with incoming data (see PrependBuf() sample). TIAPI will then UnprependBuf() it's header if it had added one, and then send the buffer to JSP. JSP will UnprependBuf() it's header and send the buffer to JIP. As well, JIP will UnprependBuf() the information it had added, and send it up to AM. Finally, AM will pass the buffer to the Device.

Assume the device wants to copy the data into a non-Tbcb buffer and without the padding. It would do so as follows:

```
memcpy( devicebuffer, BufDataAddr(thisbuff),
BufDataLen(thisbuff) );
```

Identity: BufFreeSpace

Description: Returns the number of bytes available in the data portion of the Tbcb.

Detail:
```
Tu32 BufFreeSpace( Tbcb* prmTbcbPtr );
```

Relevant files: tops.h
appl_buf.h

Parameters: The parameter prmTbcbPtr is a pointer to a Tbcb.

Return values:
This function returns the number of bytes available in the data portion of the buffer.

Usage: This is used by the buffer functions for verifying that there is enough room to do the requested function calls. It is also used when it is desired to determine if there is enough room to prepend data into the buffer.

Requirements: The buffer must be created with CreateBuf().

See also: CreateBuf
MustCreateBuf
DestroyBuf
ReinitializeBuf
PrependCopyToBuf
PrependBuf
UnprependBuf
BufDataSpace
BufDataLen
BufDataAddr

Samples: See PrependBuf() sample.

Identity: BufDataAddr

Description: Returns a pointer to the data in the buffer.

Detail:
```
void* BufDataAddr( Tbcb* prmTbcbPtr );
```

Relevant files: tops.h
appl_buf.h

Parameters: The parameter prmTbcbPtr is a pointer to a Tbcb.

Return values:
This function returns a void pointer to the data in the buffer.

Usage: This is used whenever it is desired to view the data in the buffer.

Requirements: The buffer must have been created with CreateBuf().

See also: CreateBuf
MustCreateBuf
DestroyBuf
ReinitializeBuf
PrependCopyToBuf
PrependBuf
UnprependBuf
BufDataSpace
BufDataLen
BufFreeSpace

Samples: See UnprependBuf() sample.

Appendix I

Queue Function Reference

Part of System API

Identity: QueueInitialize

Description: Initialize a TqHead.

Detail:
```
void QueueInitialize( TqHead * prmHeadPtr );
```

Relevant files: appl_queue.h

Parameters:
Return values: The parameter prmHeadPtr is a pointer to the TqHead belonging to the Queue upon which this function's operation should be performed.

Usage: This function should be used when it is desired a queue be created.

Requirements: No requirements.

See also: All Queue* functions.

Samples:
```
TqHead MyQueue;

QueueInitialize(&MyQueue);
/* Now MyQueue is ready to be manipulated */
```

Identity: QueueSize

Description: Returns a count of the number of nodes in the Queue.

Detail:
```
Tu32 QueueSize( TqHead * prmHeadPtr );
```

Relevant files: appl_queue.h

Parameters : The parameter prmHeadPtr is a pointer to the TqHead belonging to the Queue upon which this function's operation should be performed.

Return values: The function returns the number of elements in the Queue.

Usage: This should be used to check the size of the queue.

Requirements: Prior to this function call, the queue must have been initialized with a call to QueueInitialize().

See also: See Queue* functions.

Samples:
```
TqHead MyQueue;

/* Assume it has already been initialized */

void thisFunction ( void )
{
Tu32 queueSize;

queueSize=QueueSize(&MyQueue);

printf("The size of MyQueue is [%ld]\n",queueSize);
}
```

Identity: QueueIsEmpty

Description: Returns TRUE if the specified Queue is empty.

Detail:
```
Tu8 QueueIsEmpty( TqHead * prmHeadPtr );
```

Relevant files: appl_queue.h

Parameters: The parameter prmHeadPtr is a pointer to the TqHead belonging to the Queue upon which this function's operation should be performed.

Return values: This function returns TRUE (1) if the Queue is empty, FALSE (0) otherwise.

Usage: This should be used to check the status of the queue.

Requirements: Prior to this call, the queue must have been initialized with a call to QueueInitialize().

See also: All Queue* functions.

Samples:
```
TqHead MyQueue;

/* Assume the queue has been previously initialized */
.
.
if ( !QueueIsEmpty(&MyQueue) )
printf("MyQueue has [%ld] elements in it\n",QueueSize(&MyQueue));
else
printf("MyQueue is EMPTY\n");
```

Identity: QueueAddToRear

Description: Add a node to the end of the Queue and increment the Queue node count.

Detail:
```
void QueueAddToRear( TqHead * prmHeadPtr, void * prmNodePtr );
```

Relevant files: appl_queue.h

Parameters: The parameter prmHeadPtr is a pointer to the TqHead belonging to the Queue upon which this function's operation should be performed.

Return values: The parameter prmNodePt is a pointer to the node that this function will add to the end of the Queue.

Usage: This is used with the Queue functions.

Requirements: Prior to this call, the queue must have been initialized with a call to QueueInitialize().

See also: All Queue* functions.

Samples:
```
TqHead MyQueue;

/* Assume the queue has been initialized and that the queue is
not empty */

void thisFunct( Tchar * someString )
{
Tchar * thisStuff;

thisStuff=(Tchar*)malloc((strlen(someString)+1)*sizeof(Tchar));
strcpy(thisStuff,someString);

QueueAddToRear(&MyQueue,(void*)thisStuff);
.
.
}
```

Identity: QueueAddToFront

Description: Add a node to the front of the Queue and increment the Queue node count.

Detail:
```
void QueueAddToFront( TqHead * prmHeadPtr, void * prmNodePtr );
```

Relevant files: appl_queue.h

Parameters:
Return values:

The parameter prmHeadPtr is a pointer to the TqHead belonging to the Queue upon which this function's operation should be performed.

The parameter prmNodePtr is a pointer to the node that this function will add to the front of the queue.

Usage: This is used with the Queue functions.

Requirements: Prior to this call, the queue must have been initialized with a call to QueueInitialize().

See also: All Queue* functions.

Samples:
```
TqHead MyQueue;

/* Assume the queue has been initialized and that the queue is
not empty */

void thisFunct( Tchar * someString )
{
Tchar * thisStuff;

thisStuff=(Tchar*)malloc((strlen(someString)+1)*sizeof(Tchar));
strcpy(thisStuff,someString);

QueueAddToFront(&MyQueue,(void*)thisStuff);
.
.
}
```

Identity: QueueRemoveFront

Description: Remove a node from the front of the Queue and return in *prmNodePtrPtr.

Detail:
```
void QueueRemoveFront( TqHead * prmHeadPtr, void** prmNodePtrPtr
);
```

Relevant files: appl_queue.h

Parameters:
Return values:

The parameter prmHeadPtr is a pointer to the TqHead belonging to the Queue upon which this function's operation should be performed.

The parameter prmNodePtrPtr is a pointer to the place where this function will put the pointer to the node it removes from the Queue.

Usage: This is used with the Queue functions.

Requirements: Prior to this call, the queue must have been initialized with a call to QueueInitialize().

See also: All Queue* functions.

Samples:
```
TqHead MyQueue;

/* Assume the queue has been initialized and that the queue is
not empty */
/* Assume someString="This is a test" */
void thisFunct( Tchar * someString )
{
Tchar * thisStuff;

thisStuff=(Tchar*)malloc((strlen(someString)+1)*sizeof(Tchar));
strcpy(thisStuff,someString);

QueueAddToFront(&MyQueue,(void*)thisStuff);
thisStuff=NULL;
QueueRemoveFront(&MyQueue,(void*)&thisStuff);
printf("The first thing on the queue is [%s]\n",thisStuff);
/* thisStuff will be "This is a test" */
}
```

Identity: QueueRemoveRear

Description: Remove a node from the rear of the Queue and return the node pointer in *prmNodePtrPtr.

Detail:
```
void QueueRemoveRear( TqHead * prmHeadPtr, void** prmNodePtrPtr
);
```

Relevant files: appl_queue.h

Parameters:
Return values:

The parameter prmHeadPtr is a pointer to the TqHead belonging to the Queue upon which this function's operation should be performed.

The parameter prmNodePtrPtr is a pointer to the place where this function will put the pointer to the node it removes from the Queue.

Usage: This is used with the Queue functions.

Requirements: Prior to this call, the queue must have been initialized with a call to QueueInitialize().

See also: All Queue* functions.

Samples:
```
TqHead MyQueue;

/* Assume the queue has been initialized and that the queue is
empty */
/* Assume someString="This is a test" */
void thisFunct( Tchar * someString )
{
Tchar * thisStuff;

thisStuff=(Tchar*)malloc((strlen(someString)+1)*sizeof(Tchar));
strcpy(thisStuff,someString);

QueueAddToFront(&MyQueue,(void*)thisStuff); /* There is only one
thing on the queue */
thisStuff=NULL;
QueueRemoveRear(&MyQueue,(void*)&thisStuff);
printf("The first/last/only thing on the queue is
[%s]\n",thisStuff);
/* thisStuff will be "This is a test" */
}
```

Identity: QueuePeekAtFront

Description: Get the pointer to the node from the front of the Queue and return it in *prmNodePtrPtr.

Detail:
```
void QueuePeekAtFront( TqHead * prmHeadPtr, void** prmNodePtrPtr
);
```

Relevant files: appl_queue.h

Parameters :
Return values:
The parameter prmHeadPtr is a pointer to the TqHead belonging to the Queue upon which this function's operation should be performed.

The parameter prmNodePtrPtr is a pointer to the place where this function will put the pointer to the first node in the Queue.

Usage: This is used with the Queue functions. It allows the user to look at the information instead of removing information from the queue.

Requirements: Prior to this call, the queue must have been initialized with a call to QueueInitialize().

See also: All Queue* functions.

Samples:
```
TqHead MyQueue;

/* Assume the queue has been initialized */
void thisFunct( void )
{
Tchar * thisStuff;

QueuePeekAtFront(&MyQueue,(void*)&thisStuff);
if ( thisStuff != NULL )
printf("The first thing in the queue is [%s]\n",thisStuff);
else
printf("The queue is empty\n");
}
```

Identity: QueuePeekAtRear

Description: Get the pointer to the node at the rear of the Queue and return it in *prmNodePtrPtr.

Detail:
```
void QueuePeekAtRear( TqHead * prmHeadPtr, void** prmNodePtrPtr
);
```

Relevant files: appl_queue.h

Parameters:
Return values:

The parameter prmHeadPtr is a pointer to the TqHead belonging to the Queue upon which this function's operation should be performed.

The parameter prmNodePtr is a pointer to the node for which the next node is returned.

The parameter prmNodePtrPtr is a pointer to the place where this function will put the pointer to the last node in the Queue.

Usage: This is used with the Queue functions; it is very useful in parsing through a queue.

Requirements: Prior to this call, the queue must have been initialized with a call to QueueInitialize(). Typically, the prmNodePtr element of the queue has been reached with one of the Queue functions.

See also: All Queue* functions.

Samples: Same as sample code for QueuePeekAtFront(). Just replace QueuePeekAtFront with QueuePeekAtRear.

Identity: QueuePeekAtNext

Description: Get the pointer to the node following the one pointed to by prmNodePtr, and return it in *prmNodePtrPtr.

Detail:
```
void QueuePeekAtNext( TqHead * prmHeadPtr, void* prmNodePtr,
void** prmNodePtrPtr );
```

Relevant files: appl_queue.h

Parameters and Return values: prmHeadPtr - A pointer to the TqHead belonging to the Queue upon which this function's operation should be performed.

The parameter prmNodePtr is a pointer to the node for which the next node is returned.

The parameter prmNodePtrPtr is a pointer to the place where the function will put the pointer to the node following the one pointed to by prmNodePtr.

Usage: This is used with the Queue functions. It is very useful in parsing through a queue.

Requirements: Prior to this call, the queue must have been initialized with a call to QueueInitialize(). Typically, the prmNodePtr element of the queue has been reached with one of the Queue functions.

See also: All Queue* functions.

Samples:
```
TqHead MyQueue;


void thisFunct( void )
{
Tchar * thisString, * nextString;

printf("First element in queue:\n");
thisString=nextString=NULL;
QueuePeekAtFront(&MyQueue,(void*)&thisString);
while ( thisString != NULL )
{
QueuePeekAtNext(&MyQueue,(void*)thisString,(void*)&nextString);
printf("Queue element is [%s]\n", thisString);
thisString=nextString;
}
printf("Last element in queue\n");
}
```

Identity: QueuePeekAtPrev

Description: Get the pointer to the node preceding the one pointed to by prmNodePtr and return it in *prmNodePtrPtr.

Detail:
```
void QueuePeekAtPrev( TqHead * prmHeadPtr, void* prmNodePtr,
void** prmNodePtrPtr );
```

Relevant files: appl_queue.h

Parameters:
Return values:

The parameter prmHeadPtr is a pointer to the TqHead belonging to the Queue upon which this function's operation should be performed.

The parameter prmNodePtr is a pointer to the node for which the previous node is returned.

The parameter prmNodePtrPtr is a pointer to the place where this function will put the pointer to the node preceding the one pointed to by prmNodePtr.

Usage: This is used with the Queue functions. It is very useful for parsing a queue starting from the end of the queue.

Requirements: Prior to this call, the queue must have been initialized with a call to QueueInitialize().

Typically, the prmNodePtr element of the queue has been reached with one of the Queue functions.

See also: All Queue* functions.

Samples:
```
TqHead MyQueue;


void thisFunct( void )
{
Tchar * thisString, * nextString;

printf("Last element in queue:\n");
thisString=nextString=NULL;
QueuePeekAtRear(&MyQueue,(void*)&thisString);
while ( thisString != NULL )
{
QueuePeekAtPrev(&MyQueue,(void*)thisString,(void*)&nextString);
printf("Queue element is [%s]\n", thisString);
thisString=nextString;
}
printf("First element in queue\n");
}
```

Identity: QueueInsertAfter

Description: Insert *prmNewNodePtr into the Queue after node *prmNodePtr.

Detail:
```
void QueueInsertAfter( TqHead * prmHeadPtr, void* prmNodePtr,
void* prmNewNodePtr );
```

Relevant files: appl_queue.h

Parameters:
Return values:

The parameter prmHeadPtr is a pointer to the TqHead belonging to the Queue upon which this function's operation should be performed.

The parameter prmNodePtr is a pointer to the node after which the new node will be inserted into the Queue.

The parameter prmNewNodePtr is a pointer to the node which will be inserted into the Queue.

Usage: This is used with the Queue functions.

Requirements: Prior to this call, the queue must have been initialized with a call to QueueInitialize().

Typically, the prmNodePtr is retrieved from the queue using one of the Queue functions. prmNodePtr should not be removed from the queue prior to this call.

See also: All Queue* functions.

Samples:

```
TqHead YourQueue;
/* Nodes of YourQueue are of type Ts16 */
/* Assume YourQueue has been initialized and nodes have been
added to it */
void insertYourQueue( Ts16 thisval, Ts16 * newnode )
{
Ts16 * qval, * nextval;

/* First find the node in the queue */
qval=nextval=NULL;
QueuePeekAtFront(&YourQueue,(void*)&qval);
while ( (qval != NULL) && (*qval != thisval) )
{
QueuePeekAtNext(&YourQueue,(void*)qval,(void*)&nextval);
qval=nextval;
}

if ( qval == NULL )
{
printf("Couldn't find the node with value [%d].  Insert newnode
at end\n",thisval);
QueueInsertRear(&YourQueue,(void*)newnode);
}
else
{
printf("Found the node with value [%d].  Insert newnode after
it.\n",*qval);
QueueInsertAfter(&YourQueue,(void*)qval,(void*)newnode);
}
}
```

Identity: QueueInsertBefore

Description: Insert *prmNewNodePtr into the Queue before the node pointed to by *prmNodePtr.

Detail:
```
void QueueInsertBefore( TqHead * prmHeadPtr, void* prmNodePtr,
void* prmNewNodePtr );
```

Relevant files: appl_queue.h

Parameters :
Return values:

The parameter prmHeadPtr is a pointer to the TqHead belonging to the Queue upon which this function's operation should be performed.

The parameter prmNodePtr is a pointer to the node before which the new node will be inserted into the Queue.

The parameter prmNewNodePtr is a pointer to the node which will be inserted into the Queue.

Usage: This is used with the Queue functions.

Requirements: Prior to this call, the queue must have been initialized with a call to QueueInitialize().

Typically, the prmNodePtr is retrieved from the queue using one of the Queue functions. prmNodePtr should not be removed from the queue prior to this call.

See also: All Queue* functions.

Samples: See QueueInsertAfter() sample code for the basic concept.

Identity: QueueRemove

Description: Remove the node *prmNodePtr from the Queue.

Detail:
```
void QueueRemove( TqHead * prmHeadPtr, void* prmNodePtr );
```

Relevant files: appl_queue.h

Parameters:
Return values:

The parameter prmHeadPtr is a pointer to the TqHead belonging to the Queue upon which this function's operation should be performed.

The parameter prmNodePtr is a pointer to the node to remove from the Queue.

Usage: This is used with the Queue functions.

Requirements: Prior to this call, the queue must have been initialized with a call to QueueInitialize().

Typically, the prmNodePtr is retrieved from the queue using one of the Queue functions.

See also: All Queue* functions.

Samples:

```
TqHead YourQueue;
/* Nodes of YourQueue are of type Ts16 */
/* Assume YourQueue has been initialized and nodes have been
added to it */
void removeFromYourQueue( Ts16 thisval )
{
Ts16 * qval, * nextval;

/* First find the node in the queue */
qval=nextval=NULL;
QueuePeekAtFront(&YourQueue,(void*)&qval);
while ( (qval != NULL) && (*qval != thisval) )
{
QueuePeekAtNext(&YourQueue,(void*)qval,(void*)&nextval);
qval=nextval;
}

if ( qval == NULL )
printf("Couldn't find the node with value [%d].\n",thisval);
else
{
printf("Found the node with value [%d]. Removing it.\n",*qval);
QueueRemove(&YourQueue,(void*)qval);
}
}
```

Identity: QueueFreeAllNodes

Description: Remove all nodes from the Queue and free them with a call to Tfree().

Detail:
```
void QueueFreeAllNodes( TqHead * prmHeadPtr );
```

Relevant files: appl_queue.h

Parameters:
Return values: The parameter prmHeadPtr is a pointer to the TqHead belonging to the Queue upon which this function's operation should be performed.

Usage: This is used with the Queue functions.

Requirements: Prior to this call, the queue must have been initialized with a call to QueueInitialize().

See also: All Queue* functions.

Samples:
```
TqHead YourQueue;

/*Assume the queue has been used throughout the life of the
code*/
/*The system has gone into a cleanup state in preparation for
shutdown */

QueueFreeAllNodes(&YourQueue);
/* The queue YourQueue still exists for use, but all of the
existing nodes have been free'd */

if ( QueueIsEmpty(&YourQueue) )
printf("The queue has been cleaned out\n");  /* This will happen
*/
else
printf("BIG TIME ERROR. \n");
```

Identity: QueueMoveFrontToRear

Description: Take a node off the front of one queue and put it onto the rear of another.

Detail:
```
void * QueueMoveFrontToRear( TqHead* prmHeadFrom, TqHead*
prmHeadTo );
```

Relevant files: appl_queue.h

Parameters:
Return values:

The parameter prmHeadFrom is a pointer to the TqHead belonging to the Queue from which this function removes the node.

The parameter prmHeadTo is a pointer to the TqHead belonging to the Queue to which this function adds the node.

This function returns a void pointer to the node that was moved.

Usage: This is used with the Queue functions.

Requirements: Prior to this call, the queues must have been initialized with a call to QueueInitialize() and the prmHeadFrom queue must not be empty.

See also: All Queue* functions.

Samples:
```
TqHead Queue1;  /* Nodes are of type Tu16 */
TqHead Queue2;  /* Nodes are of type Tu16 */

/* Assume both queues have been initialized */

{
Tu16 * thisval, * thatval, * moveval;

thisval=(Tu16*)malloc(sizeof(Tu16));
*thisval=8;
QueueAddToFront(&Queue1,(void*)&thisval);
moveval=QueueMoveFrontToRear(&Queue1, &Queue2);
QueuePeekAtRear(&Queue2,(void*)&thisval);
printf("thisval [%d]\n",*thisval);
/* (*thisval) will be 8  and  thisval will equal moveval */
QueuePeekAtFront(&Queue1,(void*)&thatval);
/* thatval will NOT equal moveval */
}
```

Identity: QueueMoveRearToFront

Description: Take a node off the rear of one queue and put it onto the front of another.

Detail:
```
void * QueueMoveRearToFront( TqHead * prmHeadFrom, TqHead *
prmHeadTo );
```

Relevant files: appl_queue.h

Parameters :
Return values:

The parameter prmHeadFrom is a pointer to the TqHead belonging to the Queue from which this function removes the node.

The parameter prmHeadTo is a pointer to the TqHead belonging to the Queue to which this function adds the node.

This function returns a void pointer to the node that was moved.

Usage: This is used with the Queue functions.

Requirements: Prior to this call, the queues must have been initialized with a call to QueueInitialize() and the prmHeadFrom queue must not be empty.

See also: All Queue* functions.

Samples: See QueueMoveFrontToRear() sample code for the same concept.

Identity: QueueMoveFrontToFront

Description: Take a node off the front of one queue and put it on the front of another.

Detail:
```
void * QueueMoveFrontToFront( TqHead * prmHeadFrom, TqHead *
prmHeadTo );
```

Relevant files: appl_queue.h

Parameters: The parameter prmHeadFrom is a pointer to the TqHead belonging to the Queue from which this function removes the node.

The parameter prmHeadTo is a pointer to the TqHead belonging to the Queue to which this function adds the node.

Return values: This function returns a void pointer to the node that was moved.

Usage: This is used with the Queue functions.

Requirements: Prior to this call, the queues must have been initialized with a call to QueueInitialize() and the prmHeadFrom queue must not be empty.

See also: All Queue* functions.

Samples: See QueueMoveFrontToRear() sample code for basic concept.

Identity: QueueMoveRearToRear

Description: Take a node off the rear of one queue an put it onto the rear of another.

Detail:
```
void * QueueMoveRearToRear( TqHead * prmHeadFrom, TqHead *
prmHeadTo );
```

Relevant files: appl_queue.h

Parameters and Return values: prmHeadFrom - A pointer to the TqHead belonging to the Queue from which this function removes the node.

prmHeadTo - A pointer to the TqHead belonging to the Queue to which this function adds the node.

void * - pointer to the node that was moved.

Usage: This is used with the Queue functions.

Requirements: Prior to this call, the queues must have been initialized with a call to QueueInitialize() and the prmHeadFrom queue must not be empty.

See also: All Queue* functions.

Samples: See QueueMoveFrontToRear() sample code for the basic concept.

Appendix J

Miscellaneous Function Reference

Identity: *applFindDomain*

Description: Returns the domain of an OMN form address.

Detail:
```
Ts16 applFindDomain( in Tchar * OMNaddress );
```

Relevant files: appl_net.h

appl_tiapi.h

Parameters : The parameter OMNaddress is a JMN form address as found in the "JetSend Multi-Network Address Reference."

Return values: This function returns the domain of the OMN form address, or -1 if invalid. The domain is an enumerated type in appl_tiapi.h (Tdomain), and could also be a gateway domain (which is assigned dynamically). Gateway domains begin at 16 and get higher.

Usage: No restrictions.

Requirements: No dependencies.

See also: None.

Samples:
```
Tchar * myAddress[]=ip:15.99.99.254##1804#3;
Ts16 thisdomain;
thisdomain=applFindDomain( myAddress );
printf("The domain of the address [%s] is [%d]\n", myAddress,
thisdomain);
/* thisdomain should be ETdomainINET ( 0 ) in this particular
case */
```

Identity: *applFindSubPort*

Description: Returns the xpt port of the OMN form address.

Detail:

```
Ts32 applFindPort( in Tchar * OMNaddress, inout Tu8 * hasToken );
```

Relevant files: appl_net.h

Parameters:
Return values:

The parameter OMNaddress must be a JMN form address as found in the "JetSend Multi-Network Address Reference."

The network hasToken can be set to TRUE if the "##" delimiter is found in the OMN form address, or FALSE otherwise.

This function returns the value of the port found in the address, or -1 if none found. The port is invalid if -1 is returned and hasToken is FALSE.

Usage: No restrictions.

Requirements: No dependencies.

See also: None.

Samples:

```
Tchar * myAddress[] = ìip:15.99.99.254##1804#3î;
Ts32 thisPort;
Tu8 token;
thisPort=applFindSubPort( myAddress, &token );
if (  thisPort == -1 )
{
if ( !token )
printf(ì  Invalid Port in address [%s]\nî, myAddress );
else
printf(ì  No port in address [%s]\nî,myAddress );
}
else
printf(ì  The address [%s] has Port [%ld]\nî, myAddress,
thisPort);
/* thisPort should be 1804 in this case */
```

Identity: *applFindSubAddress*

Description: Returns the subaddress of the OMN form address, or -1 if none is specified.

Detail:
```
Ts16 applFindSubAddress( in Tchar * OMNaddress, inout Tu8 *
hasToken );
```

Relevant files: appl_net.h

Parameters:
Return values:

The parameter OMNaddress must be an OMN form address as found in the "JetSend Multi-Network Address Reference."

The parameter hasToken can be set to TRUE if the "#" delimiter is found in the OMN form address, or FALSE otherwise.

This function returns the value of the SubAddress found in the OMN form address, or -1 if none is specified. The subaddress is invalid if -1 is returned and hasToken is FALSE.

Usage: No restrictions.

Requirements: No dependencies.

See also: None.

Samples:
```
Tchar * myAddress[] = ìip:15.62.44.144##1084#3î;
Ts16 thisSubAddr;
Tu8 token;

thisSubAddr=applFindSubAddress( myAddress, &token );

if (  thisSubAddr == -1 )
{
if ( !token )
printf(ì  Invalid SubAddr in address [%s]\nî, myAddress );
else
printf(ì  No SubAddr in address [%s]\nî,myAddress );
}
else
printf(ì  The address [%s] has SubAddr [%ld]\nî, myAddress,
thisSubAddr);
/* thisSubAddr should be 3 in this case */
```

Identity: *UniqueID*

Description: Creates a queue of integers of assigned ids. Returns a unique identifier for the requested queue. This is used by external functions to keep track of unique handles.

Detail:
```
Tu32 UniqueID( TqHead * queue, Tu32 sizeof_value, Tu8 isSigned );
```

Relevant files: tops.h

appl_sys.h

appl_log.h

Parameters: The parameter queue is a pointer to a TqHead structure defining the queue from which to assign and return a unique identifier.

The parameter sizeof_value is the size of the integer for which to assign a unique identifier.

The parameter isSigned is TRUE if the requested id is a signed integer, FALSE otherwise.

Return Functions: This function returns an unsigned long int for which the requesting function should cast to the type it desires.

Usage: This can be used by functions requiring unique handles for a multitude of purposes.

Requirements: The queue parameter must have been initialized prior to this function call.

The unique identifier must be removed from the queue via a call to FreeID() when applicable.

See also: *FreeID*

Samples:

```
TqHead RqstHandles;
TqHead OtherHandles;
Tu32 globalOther;
.
.
QueueInitialize(&RqstHandles);
QueueInitialize(&OtherHandles);
.
.
void thisFunction ( void )
{
Ts16 requestHandle;
Tu32 otherHandle;

/* A new request handle is desired */
requestHandle=UniqueID(&RqstHandles,sizeof(Ts16),TRUE);
/* Assume the requestHandle returned is 55 */

/* A new other handle is desired */
otherHandle=UniqueID(&OtherHandles,sizeof(Tu32),FALSE);
/* Assume the otherHandle returned is 98 */
.
.
globalOther=otherHandle;
.
.
/* The function is done with the requestHandle and otherHandle */
FreeID(&RqstHandles,requestHandle);
}

void thatFunction ( void )
{
/* The system is done with the otherHandle */
FreeID(&OtherHandles,globalOther);
.
.
}
```

Identity: *FreeID*

Description: Removes the identifier from the specified queue.

Detail:
```
void FreeID( TqHead * queue, Tu32 val );
```

Relevant files: tops.h
appl_sys.h
appl_log.h

Parameters: The parameter queue is a pointer to a TqHead structure defining the queue from which the identifier should be removed.

Return values: The parameter Tu32val contains the ID to be freed.

Usage: This should be used by functions desiring to remove an identifier previously assigned by UniqueID().

Requirements: The queue parameter must have been initialized prior to this function call.
The unique identifier should have been assigned by UniqueID().

See also: *UniqueID*

Samples: See *UniqueID()* sample code.

Appendix K

Transport Independent API Reference

The Transport Independent API (TIAPI) provides the Jetsend SessionProtocol (JSP) with a common API to the transport service provider(s) within the device. TIAPI is like a glue layer that attaches the SP to the device-specific, network-connectivity implementation.

The TIAPI is a service, not a protocol. It simply provides a single common API to multiple transports. The protocols used under the TIAPI are the ones defined by the transports that are installed and active on the host platform. The transport is selected by the JSP when it makes a TIAPI function call.

JetSend could have chosen an industry standard transport API such as Berkeley Sockets or TLI as the TIAPI. But neither of those fits into the JetSend core architecture. The JetSend architecture requires that all TIAPI calls be asynchronous. Also, the architecture defines a specific method for passing information from TIAPI to JSP that neither Berkeley Sockets or TLI provides. TIAPI is simpler than Berkeley Sockets and TLI. This allows JSP code to be free of Transport API complexities that may obscure the JSP's own algorithms.

The JetSend session protocol resides in the JSP. That is the entity that manages JetSend session protocol issues like routing to TOPS modules based on JSP port number. It is the user of the TIAPI.

The API consists of all the constant, type, function and event (events are called replies in this API) definitions that the JSP needs to access the transport service providers within the local host.

The semantics of processing each request are presented with each function. For clarity and simplicity, the semantics are expressed in terms of Berkeley Sockets. The semantics intentionally don't obscure the central algorithm with error and special case handling. Semantics are provided to further illuminate the functionality provided, not to be an exhaustive detailed internal design or example implementation of the functions.

TIAPI uses three well-known ports for each network domain. One is the datagram transport port over which RMTP will run (in the INET domain this is a UDP port). The second is the reliable-connection-oriented transport port (in the INET domain this is a TCP port). The third is the datagram port that JSP gateway messages are sent and received through (in the INET domain this is a UDP port).

For each domain, the well-known ports are registered with the appropriate authority. In the case of the INET domain, the ports are registered through the Internet Assigned Numbers Authority (IANA). For the SIPX domain, the numbers are registered through Novell.

For the INET domain the UDP port number is 1782, the TCP port is 1782, and the gateway port is 1803.

When TIAPI announces the availability of a local network address to JSP (via the TrplyNwkStat reply), it fills in the port field of the network address with the datagram transport port over which RMTP will run. This is the JetSend contact address that should be displayed to the user of the local host as the address of his/her JetSend device.

For multi-port boxes like JetSend to provide pass-through support for JetSend, a unique set of the three well-known transport ports is needed for each of the parallel ports on the box. TIAPI is responsible for finding out which parallel port its host is attached to (if any) and using the correct set of well-known transport ports for it.

TIAPI supports two types of communications service - message and stream.

Message service uses the Reliable Message Transport Protocol (RMTP) running over the datagram protocol (UDP, IPX, etc.) that is available in the domain.

Stream service uses the standard connection-mode, full-duplex, reliable transports such as TCP, SPX, etc.

TIAPI provides an unsolicited reply, TrplyNwkStat, that tells JSP a local network address is active or that the local network address has changed to inactive. See the description section of TrplyNwkStat to find out what is meant by local addresses becoming active or inactive.

JSP uses a TconnHdl to identify a connection. TIAPI creates the TconnHdl when a connection is established and passes it up to the JSP. From that point on, the JSP uses the TconnHdl to identify to the TIAPI the connection upon which the TIAPI should perform an operation. JSP should not know and does not care what the TconnHdl is; it should not dereference it as if it were a pointer and the actual value is meaningless to JSP.

A connection may be either shutdown gracefully or aborted. If it is shutdown gracefully, the underlying transport is asked to gracefully close the connection if it has that capability. It does not mean that TIAPI will complete all xptSend() requests before closing the connection. Any pending xptSend() requests that are being worked on by TIAPI when xptHangUp() is called will be cancelled.

The TIAPI contains functions and replies. Functions pass instructions and data into the TIAPI. Replies pass data the other way, from TIAPI to JSP. Any function that may send or receive data over the network is implemented as an asynchronous (non-blocking) call. The results of its action are returned as a corresponding reply.

Any function for which there is an asynchronous reply includes a request handle parameter. The handle can be used by JSP to match-up replies with their corresponding requests. The handle is a value set by JSP before it calls the request function. The same handle value is returned to JSP in the reply that corresponds to that request call. Some requests, such as xptListen(), may generate multiple replies, one for each connection that results from the request. In those cases, the request handle from the original request call is returned in each reply. Similar to the TconnHdl for JSP TIAPI should not care what the request handle is. It only passes it through from the function call into TIAPI to the corresponding resulting reply.

Some replies may not be the direct result of a prior request. Those replies, called unsolicited replies, will have a request handle of -1.

Identity: *xptInitialize*

Description: Initialize or startup TIAPI.

Detail:
```
TcCode xptInitialize (void);
```

Relevant files: tiapi.c, tiapi_primary.c, tiapi_module.h

Parameters: none

Returns: This function returns **ETcCodeSuccess** if initialization completed successfully.

Alternately, it returns **ETcCodeFailure** if TIAPI cannot initialize properly. If this is returned, TIAPI will not function.

Usage: This must be called before any other TIAPI functions are called. It can be called only one time, when TIAPI is started

Requirements:

See also: xptShutdown

Samples:

Identity: *xptShutdown*

Description: Shutdown TIAPI.

Detail:
```
void xptShutdown (void);
```

Relevant files: tiapi_module.h
tiapi.c
tiapi_primary.c
tiapi_utils.c

Parameters: none

Returns: none

Usage: When this call is made, all pending requests are canceled by TIAPI. All open connections are aborted. No replies are generated for the canceled requests. The JSP layer should assume that all of its outstanding TIAPI requests are canceled when this function returns. TIAPI will return no more replies through xptGetNextEvent() and will not make any further calls into JSP after xptShutdown() is called.

This should be called only one time, when TIAPI is terminated. After this call TIAPI will not function again until a xptInitialize() call is completed.

Requirements:

See also: xptInitialize

Samples:

Identity: *xptCancelRequest*

Description: Cancel a previously issued request.

Detail:

```
TcCode xptCancelRequest (TrqstHdl prmRqstHdl);
```

Relevant files: tiapi.c

tiapi_primary.c

Parameters: **prmRqstHdl** is the request handle that was passed into TIAPI by JSP in the function call that made the request(s) that JSP wants to cancel.

Returns: This function returns **ETcCodeSuccess** if the request was successfully canceled.

Alternately, it returns **ETcRequestNotFound** if TIAPI cannot find the specified request, or **ETcCodeFailure** if the request was found but cannot be canceled. This code may also be returned by a TIAPI implementation that does not support cancelling the previously issued request. (Some implementations may only cancel xptListen() requests.)

Usage: TIAPI will cancel all pending requests that have the specified prmRqstHdl value.

Requirements:

See also:

Samples:

Identity: *xptRun*

Description: Accept the thread of control from the caller.

Detail:
```
void xptRun (void);
```

Relevant files: tiapi.c

tiapi_primary.c

Parameters: none

Returns: none

Usage: TOPS is single threaded for portability reasons. One way TIAPI gets hold of the thread is through this call. When TIAPI gets the thread it can use it for whatever processing it needs to do. Typically, TIAPI would respond to this call by checking for events from the underlying transports.

When TIAPI has nothing to do, it returns the thread by returning from this function.

Requirements:

See also:

Samples:

Identity: *xptXlateOSPAddr*

Description: Translate an address that is in JMN form to an address that is in TIAPI form.

Detail:

```
TcCode xpt XlateOSPAddr (TrqstHdl prmRqstHdl,
Tchar*prmOSPAddrPtr);
```

Relevant files: tiapi.c

tiapi_primary.c

Parameters: The parameter **prmRqstHdl** is an JSP generated request handle that TIAPI will include in its reply to JSP that corresponds to this call. JSP can use the handle to match-up its calls to asynchronous functions with the replies it gets back for those functions.

The parameter **prmOSPAddrPtr** is a pointer to a zero terminated ASCII string containing the JMN form address to translate.

Returns: This function returns **ETcCodeSuccess** if the request was successfully enqueued by TIAPI. TIAPI will act upon the request and generate an synchronous reply to it.

Alternately, it returns **ETcCodeFailure** if TIAPI could not enqueue the request. This may happen if TIAPI is out of queue resources.

Usage: See Appendix L for the description of JMN addresses. This function may send or receive data over the network and so is implemented as an asynchronous (non-blocking) call. The results of its action are returned as a corresponding reply.

If the JMN form address contains a hostName, xptXlateOSPAddr() uses the appropriate Name Service to translate the address. DNS, WINS, NIS, X.500, SAP, etc.

Requirements:

See also:

Samples:

Identity: *typedef TrplyXlateOSPAddr*

Description: The reply that corresponds to and returns results from the call to xptXlateOSPAddr().

Fields: The field **mbrRplyId** identifies to the receiver which kind of reply this is. Always equal to ETrplyIdXlateOSPAddr

The field **mbrCCode** identifies the conditions under which the reply was generated. It may have the following values:

ETcCodeSuccess means the TIAPI wrote a valid address translation into mbrTIAddr.

ETcCodeRequestCancelled means this request was canceled with a call to xptCancelRequest().

ETcCodeInvalidOSPAddr is given if *prmOSPAddrPtr contains a syntactically invalid address.

ETcCodeUnsupportedDomain is given if *prmOSPAddrPtr cannot be translated because the domain it specifies is not supported.

ETcCodeInvalidHostName is given if *prmOSPAddrPtr cannot be translated because the host name it specifies cannot be resolved into a network address.

The field **mbrRqstHdl** is the request handle that was passed into TIAPI by JSP in the function call that corresponds to this reply.

The field **mbrTIAddrPtr** is the structure into which the address translation is written.

Relevant files: tiapi_utils_pub.h

tiapi_primary.c

tiapi.c

tiapi_utils.c

Usage:

Requirements:

See also:

Samples:

Identity: *xptXlateTIAddr*

Description: Translate an address that is in TIAPI form to an address that is in JMN form.

Detail:

```
TcCode xptXlateTIAddr (TrqstHdl prmRqst Hdl, Taddr *prmTIAddrP
Ts32 prmMaxBytes, Tchar *prmOSPAddrPtr);
```

Relevant files: tiapi.c

tiapi_primary.c

Parameters:

prmRqstHdl:

An JSP generated request handle that TIAPI will include in its reply to JSP that corresponds to this call. JSP can use the handle to match-up its calls to asynchronous functions with the replies it gets back for those functions.

prmTIAddrPtr:

Pointer to a Taddr containing the TIAPI address to translate.

prmMaxBytes:

The maximum number of bytes that can be written into *prmOSPAddrPtr.

prmOSPAddrPtr:

A pointer to a buffer into which the adress translation will be written.

Returns:

ETcCodeSuccess:

If the request was successfully enqueued by TIAPI. TIAPI will act upon the request and generate an asynchronous reply to it.

ETcCodeFailure:

TIAPI could not enqueue the request. This may happen if TIAPI is out of queue resources.

Usage: See "JetSend Multi-Network Address Reference" for the description of JMN addresses.

This function may send or receive data over the network and so is implemented as an asynchronous (non-blocking) call. The results of its action are returned as a corresponding reply.

Requirements:

See also:

Samples:

Identity: *typedef TrplyXlateTIAddr*

Description: The reply that corresponds to and returns results from the call to xptXlateTIAddr().

Fields:

mbrRplyId:

Identifies to the receiver which kind of reply this is. Always equal to: ETrplyIdXlateTIAddr

mbrCCode:

Identifies the conditions under which the reply was generated.

ETcCodeSuccess:

TIAPI wrote a valid address translation into *mbrOSPAddrPtr.

ETcCodeRequestCancelled:

This request was canceled with a call to xptCancelRequest().

ETcCodeInvalidTIAddr:

If *prmTIAddrPtr contains an invalid address.

ETcCodeUnsupportedDomain:

If *prmTIAddrPtr cannot be translated because the domain it specifies is not supported.

mbrRqstHdl:

The request handle that was passed into TIAPI by TI in the function call that corresponds to this reply.

mbrOSPAddrPtr:

The same Tchar* that was passed into TIAPI via the corresponding call to xptXlateTIAddr(). A pointer to a buffer containing the translated address. The JMN form address is stored as a zero terminated ASCII string.

Relevant files:

Parameters:

Returns:

Usage:

Requirements:

See also:

Samples:

Identity: *typedef TrplyNwkStat*

Description: This is an unsolicited reply that TIAPI generates anytime the status of one of the underlying networks changes. When TIAPI first detects that a local address is available it sends a TrplyNwkStatus to JSP notifying JSP that it can use that address to establish connections. Likewise, when TIAPI detects that an address is no longer available it notifies JSP of that event also.

A single JetSend device may be located at multiple network addresses. It may be located at addresses in multiple domains as well as at multiple addresses within a single domain. As an example, a JetSend device with two network interfaces could be found at all of the following addresses simultaneously:

Interface 1:

SIPX address: 12 34 56 78 90 AB 12 34 56 78

INET address: 15.99.99.253

Interface 2:

INET address: 15.99.99.254

One card has IPX and IP enabled while the second card has only IP enabled.

These address may change while JetSend is running. Using the example above, the network administrator may do any of the following while JetSend is running:

- Disable one or more of the addresses.
- Enable one or more new addresses (such as an IPX address on the second card).
- Change one or more of the addresses.

In the example above, if all three addresses are initially active when JetSend starts running, TIAPI will generate three TrplyNwkStatus events for JSP - one for each address. If at a later time the administrator disables INET address 15.99.99.254, TIAPI will send a new TrplyNwkStatus for that address to JSP telling it that the address is no longer available. If the administrator changes INET address 15.99.99.253 to 15.99.99.250, TIAPI will send a TrplyNwkStatus for 15.99.99.253 telling JSP it is no longer available. Then TIAPI will send another TrplyNwkStatus, this one for 15.99.99.250, telling JSP it is available.

If JSP has connections open through an address when the address is disabled or changed, JSP does not have to take any special action. Those connections will simply fail and JSP will receive a TrplyDisconnect just as it would if the connection failed for any other reason.

Fields:

mbrRplyId:

Identifies to the receiver which kind of reply this is. Always equal to: ETrplyIdNwkStat

mbrCCode:

Identifies the conditions under which the reply was generated. Always ETcCodeSuccess.

mbrRqstHdl:

No meaning, always set to -1 as are all unsolicited replies.

mbrLocalAddress:

The network domain and network address about which TIAPI is returning status. The mbrPort field (or mbrSocket field in SIPX addresses) is set to the well-known RMTP port within the domain. If the JetSend appliance is attached to a parallel port on a multi-port print server, TIAPI will compute the correct well-known RMTP port for that parallel port.

mbrStatus:

The status - see typedef TnwkStat for possible values.

Relevant files:

Parameters:

Returns:

Usage:

Requirements:

See also:

Samples:

Identity: *xptSendDatagram()*

Description: Send a buffer of data through as a datagram.

This function may send or receive data over the network and so is implemented as an asynchronous (non-blocking) call. The results of its action are returned as a corresponding reply.

Detail:

Relevant files:

Parameters:

prmRqstHdl:

An JSP generated request handle that TIAPI will include in its reply to JSP that corresponds to this call. JSP can use the handle to match-up its calls to asynchronous functions with the replies it gets back for those functions.

prmLocalAddrPtr:

A pointer to a TIAPI address structure containing the address of the local host through which TIAPI should attempt to send the datagram. If prmLocalAddressPtr is NULL, TIAPI will select the local address through which it will attempt to send the datagram (same behavior as using INADDR_ANY in BerkeleySockets). If it is not NULL, TIAPI will use the local address it points to when attempting to send the datagram.

- If the port field (or socket field in the case of SIPX domain) in the address is zero, TIAPI will send through its well-known datagram port. If prmLocalAddrPtr is NULL, the port field is treated as if it were set to zero.
- Else if the port field contains one of the well-known RMTP port numbers, TIAPI will compute the corresponding well-known datagram port and send through that port.
- Otherwise, TIAPI will open the specified port and send through it.

prmForeignAddrPtr:

A pointer to a TIAPI address structure containing the address of the foreign peer with which TIAPI should attempt to send the datagram.

- If the port field (or socket field in the case of SIPX domain) in the address is zero, TIAPI will send to the default well-known datagram port number.
- Else if the port field contains one of the well-known RMTP port numbers, TIAPI will compute the corresponding well-known datagram port and send to that port.
- Otherwise, TIAPI will send to the specified port.

prmDataPtr:

A pointer to a buffer control block that xptSendDatagram() will use to send data. xptSendDatagram() will assume that the payload portion of the Tbcb is large enough to send prmMaxBytes of data from. TIAPI will simply send data from the buffer at the address returned by calling BufDataAddr() on the Tbcb.

prmBytesToSend:

The length of the data pointed to by prmDataPtr.

Returns:

ETcCodeSuccess:

If the request was successfully enqueued by TIAPI. TIAPI will act upon the request and generate an asynchronous reply to it.

ETcCodeFailure:

TIAPI could not enqueue the request. This may happen if TIAPI is out of queue resources.

Usage:

Requirements:

See also:

Samples:

Identity: *typedef TrplySendDatagram*

Description: The reply that corresponds to and returns results from the call to xptSendDatagram().

Fields:

mbrRplyId:

Identifies to the receiver which kind of reply this is. Always equal to: ETrplyIdSendDatagram

mbrCCode:

Identifies the conditions under which the reply was generated.

ETcCodeSuccess:

The data was sent.

ETcCodeFailure:

Internal resources are unavailable to complete the request.

ETcCodeRequestCancelled:

This request was canceled with a call to xptCancelRequest().

ETcCodeTooLarge:

The data is larger than the underlying transport will support.

ETcCodeInvalidLocalPort:

TIAPI cannot use the specified local port for this request.

ETcCodeUnsupportedDomain:

The domain specified in the address is not supported.

ETcCodeInvalidConn:

The connection is invalid.

ETcCodeAbort:

It failed for another reason.

mbrRqstHdl:

The request handle that was passed into TIAPI by JSP in the function call that corresponds to this reply.

mbrLocalAddr:

A structure containing the local address through which TIAPI sent the datagram.

mbrForeignAddr:

A structure containing the address of the foreign peer to which TIAPI sent the datagram.

mbrDataPtr:

The same Tbcb* that was passed into TIAPI via the corresponding call to xptSendDatagram(). A pointer to a buffer containing the data that was transmitted.

mbrBytesSent:

The number of bytes actually transmitted.

Relevant files:

Parameters:

Returns:

Usage:

Requirements:

See also:

Samples:

Identity: *xptReceiveDatagram*

Description: Accept a buffer of data from a datagram. This function passes a buffer into TIAPI that TIAPI can use for receiving a datagram.

This function may send or receive data over the network and so is implemented as an asynchronous (non-blocking) call. The results of its action are returned as a corresponding reply.

Detail:

Relevant files:

Parameters:

prmRqstHdl:

An JSP generated request handle that TIAPI will include in its reply to JSP that corresponds to this call. JSP can use the handle to match-up its calls to asynchronous functions with the replies it gets back for those functions.

prmLocalAddrPtr:

A pointer to a TIAPI address structure containing the address of the local host over which TIAPI should listen for a datagram.

- If the port field (or socket field in the case of SIPX domain) in the address is zero, TIAPI will listen through its well-known datagram port.
- Else if the port field contains one of the well-known RMTP port numbers, TIAPI will compute the corresponding well-known datagram port and listen through that port.
- Otherwise, TIAPI will open the specified port and listen through it.

prmMaxBytes:

The maximum number of bytes that can be written into *prmDataPtr.

prmDataPtr:

A pointer to a buffer control block that xptReceiveDatagram() will use to receive data. xptReceiveDatagram() will assume that the payload portion of the Tbcb is large enough to receive prmMaxBytes of data. Prior to calling xptReceiveDatagram(), JSP will have called PrependBuf() to make room in the Tbcb for prmMaxBytes of data. TIAPI will simply receive data into the buffer at the address returned by calling BufDataAddr() on the Tbcb.

Returns:

ETcCodeSuccess:

If the request was successfully enqueued by TIAPI. TIAPI will act upon the request and generate an asynchronous reply to it.

ETcCodeFailure:

TIAPI could not enqueue the request. This may happen if TIAPI is out of queue resources.

Usage:

Requirements:

See also:

Samples:

Identity: *typedef TrplyReceiveDatagram*

Description: The reply that corresponds to and returns results from the call to xptReceiveDatagram().

Fields:

mbrRplyId:

Identifies to the receiver which kind of reply this is. Always equal to: ETrplyIdReceiveDatagram

mbrCCode:

Identifies the conditions under which the reply was generated.

ETcCodeSuccess:

Data was received.

ETcCodeFailure:

Internal resources are unavailable to complete the request.

ETcCodeRequestCancelled:

This request was canceled with a call to xptCancelRequest().

ETcCodeInvalidLocalPort:

TIAPI cannot use the specified local port for this request.

ETcCodeUnsupportedDomain:

The domain specified in the address is not supported.

ETcCodeInvalidConn:

The connection is invalid.

ETcCodeAbort:

It failed for another reason.

mbrRqstHdl:

The request handle that was passed into TIAPI by JSP in the function call that corresponds to this reply.

mbrLocalAddr:

A structure containing the local address through which TIAPI received the datagram.

mbrForeignAddr:

A structure containing the address of the foreign peer from which TIAPI has received a datagram.

mbrDataPtr:

The same Tbcb* that was passed into TIAPI via the corresponding call to xptReceiveDatagram(). A pointer to a buffer containing the data that was received.

mbrBytesReceived:

The number of bytes received.

Relevant files:

Parameters:

Returns:

Usage:

Requirements:

See also:

Samples:

Identity: *xptCall*

Description: Attempt to open a connection to another device.

This function may send or receive data over the network and so is implemented as an asynchronous (non-blocking) call. The results of its action are returned as a corresponding reply.

A platform that is multi-homed is one that has more than one local address within the same domain. For example, a LaserJet with two netowrk interfaces can have two IP addresses. In some cases it may be possible to connect to a foreign peer through one of those local addresses but not the other. If prmLocalAddrPtr is NULL, xptCall() will select the local address through which it can establish a connection to the foreign peer, if any.

Detail:

Relevant files:

Parameters:

prmRqstHdl:

An JSP generated request handle that TIAPI will include in its reply to JSP that corresponds to this call. JSP can use the handle to match-up its calls to asynchronous functions with the replies it gets back for those functions.

prmServiceType:

Indicates the type of connection desired - STREAM or MESSAGE.

prmLocalAddrPtr:

A pointer to a TIAPI address structure containing the address of the local host through which TIAPI should attempt to establish a connection. If prmLocalAddressPtr is NULL, TIAPI will select the local address through which it will ,attempt to establish the connection (same behavior as using INADDR_ANY in BerkeleySockets). If it is not NULL, TIAPI will use the local address it points to when attempting to establish the connection.

- If the request is for a MESSAGE connection and...
- If the port field (or socket field in the case of SIPX domain) in the address is zero, TIAPI will call through its well-known RMTP port. If prmLocalAddrPtr is NULL, the port field is treated as if it were set to zero.
- Otherwise, TIAPI will open the specified port and call through it.
- If the request is for a STREAM connection and...
- If the port field (or socket field in the case of SIPX domain) in the address is zero, TIAPI will call through its well-known reliable-connection-oriented port. If prm-LocalAddrPtr is NULL, the port field is treated as if it were set to zero.
- Else if the port field contains one of the well-known RMTP port numbers, TIAPI will compute the corresponding well-known reliable-connection-oriented port and call through that port.
- Otherwise, TIAPI will open the specified port and call through it.

prmForeignAddrPtr:

A pointer to a TIAPI address structure containing the address of the foreign peer with which TIAPI should attempt to establish a connection.

- If the request is for a MESSAGE connection and...
- If the port field (or socket field in the case of SIPX domain) in the address is zero, TIAPI will call to the well-known RMTP port.
- Otherwise, TIAPI will call to the specified port.
- If the request is for a STREAM connection and...
- If the port field (or socket field in the case of SIPX domain) in the address is zero, TIAPI will call to the well-known reliable-connection-oriented port.
- Else if the port field contains one of the well-known RMTP port numbers, TIAPI will compute the corresponding well-known reliable-connection-oriented port and call to that port.
- Otherwise, TIAPI will call to the specified port.

Returns:

ETcCodeSuccess:

If the request was successfully enqueued by TIAPI. TIAPI will act upon the request and generate an asynchronous reply to it.

ETcCodeFailure:

TIAPI could not enqueue the request. This may happen if TIAPI is out of queue resources.

Usage:

```
lclSkt  = socket(prmLocalAddrPtr->mbrDomain,STREAM,0);
bind(lclSkt,&prmLocalAddrPtr-
>mbrAddress.mbrINETAddr,lclAddrSize);
connect(lclSkt, &lclRply->mbrForeignAddr-
>mbrAddress.mbrINETAddr, &lclAddrSize);
```

Requirements:

See also:

Samples:

Identity: *typedef TrplyActiveConnect*

Description: The reply that corresponds to and returns results from the call to xptCall().

Fields:

mbrRplyId:

Identifies to the receiver which kind of reply this is. Always equal to: ETrplyIdActiveConnect

mbrCCode:

Identifies the conditions under which the reply was generated.

ETcCodeSuccess:

A connection is established and ready to use.

ETcCodeFailure:

Internal resources are unavailable to complete the request.

ETcCodeRequestCancelled:

This request was canceled with a call to xptCancelRequest().

ETcCodeUnsupportedDomain:

The domain specified in the address is not supported.

ETcCodeInvalidAddr:

The specified address is syntactically invalid.

ETcCodeAddrNotAvail:

The specified address is not available from the local host.

ETcCodeBusy:

The attempt to connect was refused.

ETcCodeNoAnswer:

The attempt to connect timed-out.

ETcCodeInvalidLocalPort:

TIAPI cannot use the specified local port for this request.

ETcCodeAbort:

It failed for another reason.

mbrRqstHdl:

The request handle that was passed into TIAPI by JSP in the function call that corresponds to this reply.

mbrLocalAddr:

A structure containing the local address through which TIAPI has established a connection.

mbrForeignAddr:

A structure containing the address of the foreign peer with which TIAPI has established a connection.

mbrConnHdl:

The identifier that JSP will use in all its interactions with TIAPI to refer to this connection.

mbrMaxMsgSize:

The maximum size of a single JSP message (including the JSP header) that the transport can send on this connection.

Relevant files:

Parameters:

Returns:

Usage:

Requirements:

See also:

Samples:

Identity: *xptListen*

Description: Accept connection requests from other devices.

This function may send or receive data over the network and so is implemented as an asynchronous (non-blocking) call. The results of its action are returned as a corresponding reply.

Detail:

Relevant files:

Parameters:

prmRqstHdl:

An JSP generated request handle that TIAPI will include in its reply to JSP that corresponds to this call. JSP can use the handle to match-up its calls to asynchronous functions with the replies it gets back for those functions.

prmServiceType:

Indicates the type of connections to listen for - STREAM or MESSAGE.

prmLocalAddrPtr:

A pointer to a TIAPI address structure containing the address of the local host over which TIAPI should listen for connection attempts.

- If the request is for a MESSAGE connection and...
- If the port field (or socket field in the case of SIPX domain) in the address is zero, TIAPI will listen through its well-known RMTP port.
- Otherwise, TIAPI will open the specified port and listen through it.
- If the request is for a STREAM connection and...
- If the port field (or socket field in the case of SIPX domain) in the address is zero, TIAPI will listen through its well-known reliable-connection-oriented port.
- Else if the port field contains one of the well-known RMTP port numbers, TIAPI will compute the corresponding well-known reliable-connection-oriented port and listen through that port.
- Otherwise, TIAPI will open the specified port and listen through it.

Returns:

ETcCodeSuccess:

If the request was successfully enqueued by TIAPI. TIAPI will act upon the request and generate asynchronous replies to it.

ETcCodeFailure:

TIAPI could not enqueue the request. This may happen if TIAPI is out of queue resources.

Usage:

```
lclSkt  = socket(prmLocalAddrPtr->mbrDomain,STREAM,0);
bind(lclSkt,&prmLocalAddrPtr-
>mbrAddress.mbrINETAddr,lclAddrSize);
listen(lclSkt,2);
while (1) {
lclRply = malloc(sizeof(TrplyPassiveConnect));
lclRply->mbrRplyId  = ETrplyIdPassiveConnect;
lclRply->mbrRqstHdl = prmRqstHdl;
lclSize = sizeof(lclRply->mbrForeignAddr);
lclNewSkt = accept(lclSkt, &lclRply->mbrForeignAddr, &lclSize);
lclRply->mbrConnHdl = //a unique connection identifier//
lclRply->mbrCCode = // //
// put lclRply on queue from which JSP retrieves events and
replies.//
}
```

Requirements:

See also:

Samples:

Identity: *typedef TrplyPassiveConnect*

Description: The reply that corresponds to and returns results from the call to xptListen().

For reasons of consistency of usage and to cede to JSP control over when and whether connections within a particular domain (like IrDA) are accepted, JSP must have called xptListen() on the domain before a connection can be accepted in that domain.

Fields:

mbrRplyId:

Identifies to the receiver which kind of reply this is. Always equal to: ETrplyIdPassiveConnect

mbrCCode:

Identifies the conditions under which the reply was generated.

ETcCodeSuccess:

A connection is established and ready to use.

ETcCodeFailure:

Internal resources are unavailable to complete the request.

ETcCodeRequestCancelled:

This request was canceled with a call to xptCancelRequest().

ETcCodeUnsupportedDomain:

The domain specified in the address is not supported.

ETcCodeInvalidAddr:

The specified address is syntactically invalid.

ETcCodeAddrNotAvail:

The specified address is not available from the local host.

ETcCodeListenAlreadyActive:

TIAPI is already listening for connection requests on the specified local address.

ETcCodeInvalidLocalPort:

TIAPI cannot use the specified local port for this request.

ETcCodeNoAnswer:

TIAPI could not connect to the specified foreign address.

ETcCodeAbort:

It failed for another reason.

mbrRqstHdl:

The request handle that was passed into TIAPI by JSP in the function call that corresponds to this reply.

mbrLocalAddr:

A structure containing the local address through which TIAPI has established a connection.

mbrForeignAddr:

A structure containing the address of the foreign peer with which TIAPI has established a connection.

mbrConnHdl:

The identifier that JSP will use in all its interactions with TIAPI to refer to this connection.

mbrMaxMsgSize:

The maximum size of a single JSP message (including the JSP header) that the transport can send on this connection.

Relevant files:

Parameters:

Returns:

Usage:

Requirements:

See also:

Samples:

Identity: *xptHangUp*

Description:

Close a connection to another device.

This request cannot be canceled.

This function may send or receive data over the network and so is implemented as an asynchronous (non-blocking) call. The results of its action are returned as a corresponding reply.

Detail:

Relevant files:

Parameters:

prmRqstHdl:

An JSP generated request handle that TIAPI will include in its reply to JSP that corresponds to this call. JSP can use the handle to match-up its calls to asynchronous functions with the replies it gets back for those functions.

prmConnHdl:

Identifier, previously generated and returned by TIAPI, that tells TIAPI the connection upon which it should perform this operation.

prmHow:

Selects whether the connection should terminate gracefully or be aborted immediately.

Returns:

ETcCodeSuccess:

If the request was successfully enqueued by TIAPI. TIAPI will act upon the request and generate an asynchronous reply to it.

ETcCodeFailure:

TIAPI could not enqueue the request. This may happen if TIAPI is out of queue resources.

ETcCodeInvalidConn:

The specified connection handle is invalid.

Usage:

```
lclSkt = //socket associated with connection//
shutdown(lclSkt,prmHow);
close(lclSkt);
```

Requirements:

See also:

Samples:

Identity: *typedef TrplyDisconnect*

Description: The reply that corresponds to and returns results from the call to xptHangUp(). It may also be generated in response to a connection failure or a request by the foreign peer to hang up. (JSP must always be prepared to receive one of these replies on an open connection even if JSP has not issued a xptHangUp() call.)

Fields:

mbrRplyId:

Identifies to the receiver which kind of reply this is. Always equal to: ETrplyIdDisconnect

mbrCCode:

Identifies the conditions under which the reply was generated.

ETcCodeSuccess:

The connection closed as gracefully as the underlying transport allows. The connection is no longer usable.

ETcCodeAbort:

The connection closed less gracefully than the underlying transport allows. The connection is no longer usable.

ETcCodeInvalidConn:

The specified connection handle is invalid.

ETcCodeConnClosed:

The connection was closed by the foreign peer. The connection is no longer usable.

ETcCodeConnFailed:

The connection failed due to a time out or other failure. The connection is no longer usable.

mbrRqstHdl:

The request handle that was passed into TIAPI by JSP in the function call that corresponds to this reply. The value is -1 if this reply was generated in response to a connection failure or a request by the foreign peer to hang up (if the value of mbrCCode is ETcCodeConnFailed or ETcCodeConnClosed).

mbrConnHdl:

Identifies which connection was closed.

Relevant files:

Parameters:

Returns:

Usage:

Requirements:

See also:

Samples:

Identity: *xptSend*

Description: Send a buffer of data through a connection to another device.

This function may send or receive data over the network and so is implemented as an asynchronous (non-blocking) call. The results of its action are returned as a corresponding reply.

Detail:

Relevant files:

Parameters:

prmRqstHdl:

An JSP generated request handle that TIAPI will include in its reply to JSP that corresponds to this call. JSP can use the handle to match-up its calls to asynchronous functions with the replies it gets back for those functions.

prmConnHdl:

Identifier, previously generated and returned by TIAPI, that tells TIAPI the connection upon which it should perform this operation.

prmDataPtr:

A pointer to a buffer control block that xptSend() will use to send data. xptSend() will assume that the payload portion of the Tbcb is large enough to send prmMaxBytes of data from. TIAPI will simply send data from the buffer at the address returned by calling BufDataAddr() on the Tbcb.

prmBytesToSend:

The length of the data pointed to by prmDataPtr.

Returns:

ETcCodeSuccess:

If the request was successfully enqueued by TIAPI. TIAPI will act upon the request and generate an asynchronous reply to it.

ETcCodeFailure:

TIAPI could not enqueue the request. This may happen if TIAPI is out of queue resources.

Note: **If TIAPI cannot send the packet due to an error with the connection, the connection should be closed down and an ETrplyIdDisconnect event should be created (with an mbrRqstHdl of -1).

ETcCodeInvalidConn:

The specified connection handle is invalid.

Usage:

```
lclSkt = //socket associated with connection//
write(lclSkt, BufDataAddr(prmDataPtr), prmBytesToSend);
```

Requirements:

See also:

Samples:

Identity: *typedef TrplySend*

Description: The reply that corresponds to and returns results from the call to xptSend().

Fields:

mbrRplyId:

Identifies to the receiver which kind of reply this is. Always equal to: ETrplyIdSend

mbrCCode:

Identifies the conditions under which the reply was generated.

ETcCodeSuccess:

The data was sent.

ETcCodeFailure:

Internal resources are unavailable to complete the request. This may mean the connection has failed. In that case TIAPI will generate a TrplyDisconnect for the connection.

ETcCodeRequestCancelled:

This request was canceled with a call to xptCancelRequest().

ETcCodeTooLarge:

The data is larger than the underlying transport will support.

ETcCodeInvalidConn:

The specified connection handle is invalid.

ETcCodeAbort:

It failed for another reason.

mbrRqstHdl:

The request handle that was passed into TIAPI by JSP in the function call that corresponds to this reply.

mbrConnHdl:

Identifies which connection data was transmitted over.

mbrDataPtr:

The same Tbcb* that was passed into TIAPI via the corresponding call to xptSend(). A pointer to a buffer containing the data that was transmitted.

mbrBytesSent:

The number of bytes actually transmitted.

Relevant files:

Parameters:

Returns:

Usage:

Requirements:

See also:

Samples:

Identity: *xptReceive*

Description: Accept a buffer of data through a connection to another device. This function passes buffers into TIAPI that TIAPI can use for receiving data over a connection.

This function may send or receive data over the network and so is implemented as an asynchronous (non-blocking) call. The results of its action are returned as a corresponding reply.

Detail:

Relevant files:

Parameters:

prmRqstHdl:

An JSP generated request handle that TIAPI will include in its reply to JSP that corresponds to this call. JSP can use the handle to match-up its calls to asynchronous functions with the replies it gets back for those functions.

prmConnHdl:

Identifier, previously generated and returned by TIAPI, that tells TIAPI the connection upon which it should perform this operation.

prmMaxBytes:

The maximum number of bytes that can be written into *prmDataPtr.

prmDataPtr:

A pointer to a buffer control block that xptReceive() will use to receive data. xptReceive() will assume that the payload portion of the Tbcb is large enough to receive prmMaxBytes of data. Prior to calling xptReceive(), JSP will have called PrependBuf() to make room in the Tbcb for prmMaxBytes of data. TIAPI will simply receive data into the buffer at the address returned by calling BufDataAddr() on the Tbcb. If it is a request to receive from a MESSAGE connection (RMTP), JSP must prepend additional space in the buffer to accomadate the RMTP header.

Returns:

ETcCodeSuccess:

If the request was successfully enqueued by TIAPI. TIAPI will act upon the request and generate an asynchronous reply to it.

ETcCodeFailure:

TIAPI could not enqueue the request. This may happen if TIAPI is out of queue resources.

ETcCodeInvalidConn:

The specified connection handle is invalid.

Usage: JSP will call xptReceive() with code similar to this:

```
lclDataPtr = CreateBuf(lclMaxBytes);
PrependBuf(lclDataPtr,lclMaxBytes);
xptReceive(lclRqstHdl,lclConnHdl,lclMaxBytes,lclDataPtr);

and xptReceive will process the call with code similar to this:
lclSkt = //socket associated with connection//
read(lclSkt,BufDataAddr(prmDataPtr),prmMaxBytes);
```

Requirements:

See also:

Samples:

Identity: *typedef TrplyReceive*

Description: The reply that corresponds to and returns results from the call to xptReceive().

Fields:

mbrRplyId:

Identifies to the receiver which kind of reply this is. Always equal to: ETrplyIdReceive

mbrCCode:

Identifies the conditions under which the reply was generated.

ETcCodeSuccess:

Data was received.

ETcCodeFailure:

Internal resources are unavailable to complete the request. This may mean the connection has failed. In that case TIAPI will generate a TrplyDisconnect for the connection.

ETcCodeRequestCancelled:

This request was canceled with a call to xptCancelRequest().

ETcCodeInvalidConn:

The specified connection handle is invalid.

ETcCodeAbort:

It failed for another reason.

mbrRqstHdl:

The request handle that was passed into TIAPI by JSP in the function call that corresponds to this reply.

mbrConnHdl:

Identifies which connection data was received over.

mbrDataPtr:

The same Tbcb* that was passed into TIAPI via the corresponding call to xptReceive(). A pointer to a buffer containing the data that was received.

mbrBytesReceived:

The number of bytes received.

Relevant files:

Parameters:

Returns:

Usage:

Requirements:

See also:

Samples:

Identity: *typedef struct TrplyHdr*

Description: Any TIAPI reply structure can be cast to this type. When that is done the Trply structure can be examined to find out what type of reply it holds and the reply's status. These 3 fields are common to all TIAPI reply structures and occur as the first 3 fields in all of them.

The Trply.mbrHeader.mbrRplyId field identifies which TIAPI reply structure is contained in the Trply structure. When that is known, the corresponding mbr field can be referenced to examine the information contained in the reply.

Detail:

Relevant files:

Parameters:

Returns:

Usage:

Requirements:

See also:

Samples:

Identity: *xptGetNextEvent*

Description: Pass the next TIAPI reply up to the JSP.

JSP will call this function to retrieve replies from the TIAPI. TIAPI will store the replies that it generates on an internal queue and pass the next one in order up to the JSP when this function is called.

Detail:

Relevant files:

Parameters: **prmRplyPtrPtr:**

A pointer to a pointer to a reply structure. The reply structure pointer that is pointed to by prmRplyPtrPtr will be loaded with the address of the next reply from TIAPI.

Returns: **ETcCodeSuccess:**

If *prmRplyPtr contains a valid reply upon return.

ETcCodeFailure:

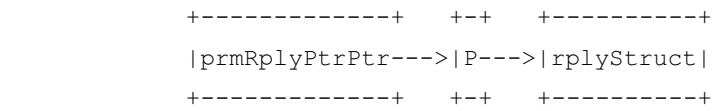
Otherwise. JSP can assum this to mean that there is no reply to retrieve at the time of the call.

Usage:

Requirements:

See also:

Samples: TIAPI owns a Trply named "rplyStruct" which it has filled in and is waiting to give to JSP. JSP owns a pointer to a Trply named "P" that is not pointing to anything. JSP passes a pointer to P to TIAPI with the call xptGetNextEvent(&P). TIAPI loads P with a pointer to rplyStruct and returns.




```
{
Trply*  P;
xptGetNextEvent(&P);
// P now contains the address of (pointer to) rplyStruct.
}
```

Identity: *xptNetworkEvent*

Description: Process a network event.

Network events are implementation defined.

This function is called by the JetSend system when a network event has occured. If xprNetworkEvent() causes any replies to be placed in the reply queue, it will call JSPNetworkEvent(void) to allow JSP to retrieve the replies.

Detail:

Relevant files:

Parameters: **prmEventPtr:**

A pointer to a device defined event message.

Returns: nothing.

Usage:

Requirements:

See also:

Samples:

Appendix L

JetSend Multi-Network Address Specification

JetSend addresses can be expressed in either of two forms: JetSend Multi-Network (JMN) or Universal Resource Locator (URL). JMN is the standard form used internally by JSP. If an address is given to JSP in URL form, JSP will store it internally in JMN form. An address in URL form implies that the type is "ip." Since JSP works internally with the more general JMN form, it translates addresses in URL form to JMN form.

Syntax

Italicized items are non-terminal; other items are literals. Items enclosed in [...] are optional fields. Other punctuation is literal.

Table 10.1:

Assignment	Value	Comment
JMN::=type: host [*##xptPort*] [*#subAddress*]		
URL::=jetsend://host[xptPort]#subAddress]		
type::=Network and/or protocol	ip ipx irda fax up lb	IP address, UDP transport IPX address, IPX transport Infrared address, TTP transport Telephone address, fax protocol Unix path loopback
host::= Network specific station name or address		15.9.99.254 myname.mycorp.com
xptPort::= Transport port number		If the xptPort is left off, it is assumed to be the default transport port number for the transport implied by the type field.
subAddress::= Sub-Address of the logical device		If this identifier is left off, it is assumed to be the remote device's default (usually 0, but not necessarily).

Examples

Table 10.2:

Example Address	Comment
ip: 15.9.99.254	JMN form
ip:myname.mycorp.com#3	JMN form with sub-address (#3)
ip:myname.mycorp.com##911#3	JMN form with port number and sub-address
fax:9.1.208396.9999	JMN form for fax number
jetsend://15.9.99.254	URL form
jetsend://myname.mycorp.com#3	URL form with sub-address
jetsend://myname.mycorp.com:1782#3	URL form with port number and sub-address

Mulit-Port Network-Parallel Gateway Addressing

A multi-port network to a parallel gateway has only one network address, but sometimes more than one parallel port. Each device connected to one of the parallel ports shares the single network address with all the other devices attached to the other parallel ports. Each parallel port has a number, beginning with 0 for the first one. JetSend associates a different “well-known” transport port with each parallel port number. consider the following UDP ports. Ports below 5000 are registered with the Internet Assigned Numbers Authority (IANA).

Table 10.3:

UDP Port (xptPort)	Associated Parallel Port	Comment
1782	0	Default
9111	1	For example only, not IANA-assigned
9112	2	For example only, not IANA-assigned

A JetSend appliance attached to parallel port 1 on a multi-port network-parallel gateway with an IP address of 15.9.99.254 would be addresses as ip: 15.9.99##9111. If that same JetSend appliance contains more than one logical JetSend device, the second logical JetSend device would be addressed as ip: 15.9.99.254##9111#1 (where 0 is the first).

If the JetSend appliance is attached to parallel port 0, it would be addressed as either ip: 15.9.99.254 or ip: 15.9.99.254##1782. The xptPort in this case can be left off, since it is the default xptPort.

Glossary

JetSend is introducing a number of new terms and new concepts that this in-depth glossary will help clarify.

Active

A sending appliance opens a session in an active way. It expects to make a connection, convey one or more surface descriptions, respond to content requests, and close the session.

(See **Passive**.)

Appliance

An appliance is a **box** that contains a device with information capability to which JetSend Communication Technology has been added. It is connected over some sort of network to other boxes. Each box views the others as a peer.

The term **appliance** emphasizes the ease-of-use, flexibility, and straightforward nature of JetSend boxes. The presence of JetSend in the box aids significantly in giving an appliance these characteristics. In the long run, peripherals are replaced with appliances.

(See **Device**, **Peripheral**, **Box**.)

Attribute-Value Pair

A fundamental pairing of a characteristic and its value. Used in combination with the encoding and surface hierarchies, the scope of the attribute is determined. A given type or class of e-material will have a uniform set of attributes and associated values.

Base Encoding

A set of surface description encoding decision paths that improves the rendering quality and fidelity of a type of e-material.

(See **Data Encoding**, **Default Encoding**, **Optional Encoding**.)

Box

The term **box** emphasizes the separate nature of a JetSend appliance, the term **physical device** emphasizes a single-function box.

The term **logical device** emphasizes the single function that applies when a box such as a PC contains more than one separable appliance.

(See **Appliance**, **Device**, **Peripheral**.)

Child Surface

Any surface in a document below the top-level surface. The term **child surface** emphasizes that the surface is only an element of a large chunk of information. Further emphasis is on the surface as a node in the tree-structure representation of the full document.

(See **Job Element**, **Sub-Surface**, **Surface**, **Top-Level Surface**.)

Class

A specific type of e-material defined to address the exchange of a specific pair of appliances. For example, the current JetSend offering is focused on scanners exchanging information with printers. Many other classes are possible.

A **class** has a specific set of attributes that it expects in a specific order. Some optional attributes may also be defined. Within the structure of the class, some attributes may offer more than one value. This gives rise to various encodings. Some encodings are mandatory, some are optional.

(See **Type**.)

Content Reply

When a content request is made by the receiving appliance, the sending appliance constructs a content reply in response.

Content Request

When a receiving appliance has a complete surface description, it requests content information (the order is based on the constraints of the sending appliance). From the content, the document is rendered.

Data Encoding

The method used to encode, encrypt, and compress surface content.

(See **Base Encoding**, **Default Encoding**, **Optional Encoding**.)

Decision Path

A set of encoding values from selection lists which together indicate an encoding of the surface. The encoding hierarchy gives (full or partial) decision paths.

Default Encoding

A set of surface description encoding decision paths always present for a type of e-material or appliance. The set of attribute value pairs is well-know for all devices that use this e-material type.

(See **Base Encoding**, **Default Encoding**, **Optional Encoding**.)

Description Reply

When a description request is made by the receiving appliance, the sending appliance constructs a description reply in response.

Description Request

Occasionally the full description of a surface does not fit into a single transaction buffer. In such a case, the receiving appliance acquires the remainder of the description by directly requesting the missing description elements.

Device

A device is a box with information capability, usually used to specify a **box** before the JetSend Communication Technology has been incorporated.

The term **device** emphasizes the functional information capabilities of boxes. It applies to peripherals before they are modified to include JetSend capability. After JetSend has been added, it applies to the part of the appliance which is separate from the JetSend negotiation layers. Device is the default term used when referring to appliances, devices, peripherals, and boxes.

(See **Appliance**, **Peripheral**, **Box**.)

Document

A complete set of surfaces spanning a body of information. A document is transmitted in a job. The term **document** emphasizes the unified nature of a collection of information chunks that can include text, graphics, and so forth. It should bring to mind a formally collated body of information.

(See **Job**, **Surface**, **Well-Known Surface**.)

Document Description Table

A four-column table that describes the structure and perhaps the content of a given document. The columns consist of surface hierarchy, encoding hierarchy, attribute, and value.

(See **Job**, **Surface**, **Well-Known Surface**.)

E-Material Description

A textual description that presents all possible encoding selections of a given document. An e-material description is created by the sending appliance. The term **e-material description** emphasizes tree-structure encodings.

(See **Impression**, **Surface Description**.)

Encoding

A single specific set of selections that a receiving appliance would request from a sending appliance resulting in a rendered, complete document on the receiving appliance. An encoding is rendered by the receiving appliance.

The term **encoding** describes a full collection of attribute/value pairs for a specific rendering of a specific surface, a selection value from a selection list, the level name in the encoding hierarchy, and one or more values from the encoding attribute list (image, text, plane, attribute, association, and file).

(See **Base Encoding**, **Data Encoding**, **Decision Path**, **Level, Default Encoding**, **Optional Encoding**, **Sub-Encoding**, **Sub-Level**.)

Encoding Hierarchy

The term **encoding hierarchy** refers to the tree-like structure of encoding choices within a single surface. The branching of the òtreeó is achieved through selection lists. Usually, a location in the hierarchy is expressed by a decision path. The sending appliance makes available any encodings enumerated in the encoding hierarchy, and the

receiving appliance selects (typically one) encoding from the tree. For example, a static image may be made available by the sending appliance at 300 by 300 pixels per inch and at 150 by 150 pixels per inch. Below this level, pixel depth may be made available at 1 or 8 bits per pixel.

(See **Decision Path**.)

Encoding Selection Process

The term **encoding selection process** refers to selection, attribute by attribute, of encoding selection values.

(See **Negotiation Process**.)

Expression

The rendering of a surface in its original form on an originating (sending) appliance.

(See **Impression**.)

Hierarchy Diagram

A diagram showing the encoding hierarchy levels and the attributes pertaining to each level.

Impression

The term **impression** emphasizes the act of transmitting the surface description from the sending appliance to the receiving appliance, and highlights the separation between the originating surface and the rendering surface. The surface description is impressed upon the surface of the receiving appliance.

(See **Expression**.)

Infrastructure

All protocol stack levels below a given level in a protocol stack. The term **infrastructure** emphasizes the encapsulation of underlying complexity below a given API between a sending and receiving appliance. For example, all of the detail below the device specific API of the sending appliance and the device specific API of the receiving appliance is simply the infrastructure. All protocol details are handled by the infrastructure.

(See **Network**.)

Job

The transmission of a document. The term **job** emphasizes the activities related to code execution and bits exchanged across the network when a document is exchanged between two appliances.

(See **Document**, **Surface**, **Well-Known Surface**.)

Job Element

The transmission of an individual surface within a document. The term **job element** emphasizes the activities related to code execution and bits exchanged across the network.

(See **Child Surface**, **Sub-Surface**, **Surface**, **Top-Level Surface**.)

Level

The term **level** refers to a single attribute block in the surface description table. All of the decision paths in the encoding hierarchy column of that table are identical.

(See **Encoding, Sub-Level**.)

Negotiation

The process whereby sending and receiving appliances determine the best mutual surface characteristics for the exchange of surface content. For efficiency, the surface description is provided by the sender in one (or more) blocks. For more complex documents, the receiving appliance makes description requests to complete the surface description. The sending appliance responds to those requests.

After the negotiation of the surface characteristics, the receiving appliance makes content requests. The sending appliance responds to those requests.

Negotiation Process

The term **negotiation process** refers to activity of a sending and a receiving appliance to identify the best mutually manipulable encoding of a surface.

(See **Encoding Selection Process**.)

Network

The term **network** emphasizes the connection between two appliances. This should bring to mind the IEEE 802.1 connection, an infrared pair, or a direct wire.

(See **Infrastructure**.)

NULL Level

The top level of an e-material description.

(See **Root Attribute Block**.)

Optional Encoding

An encoding not required for the type of e-material and/or the type of appliance in use.

(See **Base Encoding**, **Default Encoding**.)

Parameter List

A type of value field, in an attribute / value pair, which is shorthand notation for assigning several values to several attributes. For example, a reference list includes the number of child surfaces, sequencing of those surfaces, and enumeration of those surfaces within a single value field.

Passive

A receiving appliance opens a session in a passive way. It expects to listen indefinitely for push requests from sending appliances. When a request is received, it receives one or more surface descriptions, selects encodings, requests and receives content, renders documents, and returns to listen further.

(See **Active**.)

Peripheral

The term **peripheral** used in this ADK emphasizes the pre-JetSend view of boxes, the characteristics of which other boxes must be aware.

(See **Appliance**, **Device**, **Box**.)

Receiving Appliance

An appliance that primarily listens to a network awaiting requests from sending appliances. It transforms or renders e-material documents into other forms such as printed pages.

(See **Sending Appliance**.)

Reference List

The term **reference list** refers to the contents of a value field paired with a Child, ChildFront, or ChildBack attribute. This structured list contains the names of sub-surfaces to be rendered along with this surface in a complete document. Reference lists can refer to the full value field or to just the list of zero or more references within that value field.

(See **Selection List**.)

Root Attribute Block

The first level of an e-material description. The decision path at this level is NULL, since no encoding decisions have been made yet.

(See **Sub-Attribute Block**.)

Select

The act of choosing an encoding (this choice is made by a receiving appliance).

Selection

One of several values selected from a selection list. Together these choices form a decision path.

Selection List

The term **selection list** refers to the contents of a value field paired with a large number of encoding attributes. This list contains the various selection values that will then become part of a decision path for other attribute/value pairs.

(See **Reference List**.)

Selection Value

The term **selection value** emphasizes a single value taken from a selection list. This value becomes a part of a decision path at lower levels of the encoding hierarchy.

(See **Structured Value Field**, **Value**, **Value Field**.)

Sending Appliance

An appliance primarily used to push information over a network to receiving appliances. It is able to describe a document as an e-material description with one or more encodings.

(See **Receiving Appliance**.)

Session

The period of time in which two appliances are in contact. During this time, one or more jobs can be exchanged.

Appliances can exchange status information and address information. Either appliance can terminate the session.

Structured Value Field

A value field from an attribute/value pair that is not a single or simple value. Beyond a number and a string, surface descriptions allow complex values such as points, ranges, and ranges of points. Farther beyond are lists of values such as selection lists and reference lists. Still farther are parameter lists which, like parameters passed to sub-routines, impose an implied order, similar to the structure in a sub-routine argument list in programming languages.

The term **structured value field** emphasizes that some attributes take as a value a set of elements that have a certain order to them.

(See **Attribute-Value Pair**, **Selection Value**, **Value**, **Value Field**.)

Sub-Attribute Block

The lower levels of an e-material description. The decision path contains decisions about the encoding.

(See **Root Attribute Block**.)

Sub-Encoding

The term **sub-encoding** refers to the next step in the encoding selection process. At a given point in the process, an encoding has been selected as expressed in the decision path. A selection list is made available at that level. When one of the selections in the selection list is chosen, a sub-encoding is thereby selected.

(See **Encoding**, **Level**, **Sub-Level**.)

Sub-Level

The term **sub-level** refers to an attribute block in the surface description table which is below a given level.

(See **Encoding**, **Level**, **Sub-Level**.)

Sub-Surface

The next lower level in the surface hierarchy, whose name is taken from the reference list; the term **sub-surface** also emphasizes that the surface is only an element of a large chunk of information. Further emphasis is placed on the surface as a subordinate piece of the full chunk of information.

(See **Child Surface**, **Job Element**, **Surface**, **Top-Level Surface**.)

Surface

The fundamental unit that JetSend uses to model information for exchange. For example, a piece of paper is one example of a surface: Information is rendered on it, and it can contain sub-surfaces; the definition of a surface is recursive, and is the default term used among related descriptions. The term **surface** emphasizes the universal nature of information exchanged by JetSend.

Also, the term *express* emphasizes the rendering of information in the originating appliance or on an original surface; the term *impress* emphasizes the rendering of information on a remote receiving appliance or surface. The phrase *impressing a surface* articulates that information contained in an originating surface is exchanged and rendered on the receiving well-know surface. The originating repository, the information in that repository, the information in the receiving repository, and the rendering of that information are all referred to under the term **surface**, but each takes on its own sense of the term.

(See **Child Surface**, **Document**, **Job**, **Job Element**, **Sub-Surface**, **Top-Level Surface, Well-Known Surface**.)

Surface Description

The information rendered on a surface can be split into two collections. One describes the surface characteristics. For example, resolution, pixel depth, and other characteristics describe a surface, and the other collection gives the actual content of the surface. The term **surface description** emphasizes the separation between the characteristics of the surface and the contents of the surface.

A pair of JetSend appliances negotiate the best mutual surface description for a given document. Once these characteristics are selected, the actual content is exchanged. For example, a stationary bitmap image can be constrained to 300 by 300 pixels per inch with 256 level of gray (characteristics) and still give an unlimited set of actual images (content).

(See **E-Material Description**, **Impression**.)

Surface Description Table

A three-column table that describes the structure and perhaps the content of a given surface. The columns consist of encoding hierarchy, attribute, and value.

Surface Hierarchy

The tree-like structure hierarchy of surfaces that represent a given surface. The branching of the “tree” is achieved through reference lists. The sending appliance creates the structure of surfaces, and the receiving appliance interprets and renders the surfaces. A document may contain several pages which in turn may contain various text or image areas.

(See **Encoding Hierarchy**.)

Top-Level Surface

The term **top-level surface** emphasizes the surface that describes the overall structure chunk of information to be exchanged.

(See **Child Surface**, **Job Element**, **Sub-Surface**, **Surface**.)

Type

A specific class of e-material defined to address the exchange of a specific pair of appliances. For example, the current JetSend offering is focused on scanners exchanging information with printers. Many other classes are possible.

A type has a specific set of attributes that it expects in a specific order. Some optional attributes may also be defined. Within the structure of the type, some attributes may offer more than one value. This gives rise to various encodings, some mandatory, some optional.

(See **Class**.)

Value

The term **value** applies to all information in the value column of the surface description table. This corresponds to the information sent at the e-material API to construct the e-material description.

(See **Selection Value**, **Structured Value Field**, **Value Field**.)

Value Field

The term **value field** emphasizes that the value of an attribute value pair may be complex rather than simple. This complexity can be to a range, a simple list, or an order-dependent list.

(See **Selection Value**, **Structured Value Field**, **Value**.)

Well-Known Surface

The term **well-known surface** applies to eight specific information repositories. Four comprise the originating surfaces for work, status, and address information, and four comprise the receiving receptacles/repositories for work, status, and address information.

(See **Document**, **Job**, **Surface**.)

Index

Symbols

(e-material 5-1
, Sub-Encoding, Glossary-3
, Sub-Level Glossary-7
, Sub-Level. Glossary-5
, Value Field. Glossary-6
. Surface name 5-1

Numerics

8-bit binary data A-7

A

Active Glossary-1, Glossary-5
activities 5-2
address activity G-1
Address surface 5-2
Address-Receiver Activity 10-43
Address-Sender Activity 10-43
am-activity 10-42
amAddGateway C-19
amBroadcastGateway C-21
amChangeDefaultSubAddr C-23
amCreateBuf C-3
amEndSession C-10
amEnumerateNetworks C-17
amErrorEvent C-14
amFreeEvent C-6
amGetAliasAddress C-15
amGetNextEvent C-5
amInitialize C-1
amIsValidAddress C-16
amListenForSession C-12
amNetworkEvent C-18
amNewGatewayEvent C-22
amRemoveGateway C-20
amSessionEndEvent C-11
amSessionStartEvent C-8
amShutdown C-2
amStartSession C-7
applFindDomain J-1
applFindSubAddress J-3
applFindSubPort J-2
Appliance Glossary-1, Glossary-3, Glossary-6
appliance interaction 4-1
appliances 3-8
arCardReceivedEvent G-15
arInitialize G-12
arShutdown G-13
arSurfaceRejectedEvent G-14
asCardSentEvent G-8
asInitialize G-1
asSendCard G-6
asSendSelf G-10
asShutdown G-2
association 9-3
asSurfaceArrivedEvent G-3
asSurfaceGoneEvent G-5
attribute block A-3, A-8
attribute level A-1
attribute name 3-5, A-1
attribute value 3-5, A-1
attribute-value pair 3-5, 6-27, A-2, Glossary-1, Glossary-7
attributes A-2

B

Base Encoding 6-7, 6-17, A-13, Glossary-1, Glossary-2, Glossary-3, Glossary-5
big endian A-18, A-23
binary data A-24
Box Glossary-1, Glossary-3 Glossary-6
BufDataAddr H-15
BufDataLen H-12
BufDataSpace H-10
BufFreeSpace H-14

C

channels 4-9
child list A-5, A-11
child name reference A-80
Child Surface 9-2, A-5, A-11, Glossary-1, Glossary-4, Glossary-7 Glossary-8
childCount A-11, A-55, A-77
Class Glossary-2, Glossary-9
complete description 4-6
composing 3-5
compression A-28
conGetComponent D-6
conInitialize D-1
Connection Activity D-1
conPINRequestEvent D-13
conReleaseRemote Component D-9
conRemoteComponentEvent D-11
conRequestRemote Component D-7
conSendPIN D-14
conSetComponent D-4
conShutdown D-3
Contact Activity 10-43
content 4-1, A-1
Content Reply A-80, 4-6, A-1, A-23, A-50, Glossary-2
Content Request 4-6, A-1, A-21, A-37, A-49, A-63, Glossary-2
Control Flow 9-2
core 3-2
core testing 3-2
CreateBuf H-1

D

Data Encoding Glossary-1, Glossary-2, Glossary-3
decision block A-2
Decision Path 6-5, 6-13, A-3, A-10, A-13, A-14, A-19, A-36, A-61, A-78, Glossary-2,

Glossary-3
Glossary-4
decomposing 3-5, A-87
Default Encoding 4-12, 6-7, 6-17, 10-42, A-13, Glossary-1, Glossary-2, Glossary-3, Glossary-5
description 4-1, A-1
description replie 4-6, A-1, A-19, A-36, A-61, A-78, Glossary-2
description request 4-6, A-1, A-18, A-36, A-47, A-60, A-78, Glossary-2
DestroyBuf H-4
Device Glossary-1, Glossary-2, Glossary-6
Device Independent API 7-3, D-1
device-independent negotiation services 3-2
DevIndAPI 7-3
Document A-4, Glossary-3, Glossary-4, Glossary-8, Glossary-9
Document Description Table 6-9, Glossary-3

E

e, Surface Glossary-4
e, Well-Known Surface Glossary-8
e-material block 3-5, A-1
e-material data type A-7
e-material symbol A-6
electronic materia 6-1
electronic material 5-1
emAdd B-4
emAddLevel B-5
E-material 6-1, A-1
E-Material Description Glossary-3, Glossary-8
emCmp B-15
emData B-9
emFreeValue B-11
emGet B-13
emGetLevel B-14
emGetNext B-17
emInt B-8
emIntPair B-8
emKey B-6
emList B-10
emListArray B-10
emMatch B-12
emMaxData B-3
emPrepare B-1
emPrint B-18
emPrintValue B-19
emRange B-8
emRangePair B-8
emSize B-2
emString B-7
EMtype 7-1
Encoding Glossary-3, Glossary-5, Glossary-7
Encoding Hierarchy Glossary-3
encoding hierarchy 6-3, 6-6, 6-7, A-3
Encoding Hierarchy. Glossary-8
Encoding Selection Process Glossary-4
Encoding Selection Process. Glossary-5
encodings 5-1, 6-1
event 7-7
events 4-5, 5-2
Expression Glossary-4
expression 4-1
expressions 7-10

F

fil 9-3
file A-45
folder A-76
FreeID J-6
function call 5-2

G

general list A-8
guidelines A-87

H

Hierarchy Diagram Glossary-4
higher level encodings 9-3
Huffman encoding A-28

I

image 9-3
Impression 4-1, 5-5, 7-10, Glossary-3, Glossary-4, Glossary-8
In surface 5-2, E-4, E-24
in-line content A-1
inbound 7-4
inbound data 4-6
indirect child reference A-80
Infrastructure Glossary-4 Glossary-5
in-line 4-1
In-line content A-24
Interaction Testing 10-44
ISO 3166 A-9
ISO 639 A-9

J

JetSend core 3-2, 3-3
JFIF A-29
Job Glossary-3, Glossary-4, Glossary-8, Glossary-9
job activity 7-10
Job Element Glossary-2, Glossary-4, Glossary-7, Glossary-8
job ID 7-10
JPEG compression A-28, A-29

K

keywords A-2

L

languageCode A-9
Level Glossary-3, Glossary-5, Glossary-7
level name 6-7
Lists 7-4
little endian A-18, A-23
logical devices 3-8

M

memory allocation 7-3
module 5-2
MustCreateBuf H-3

N

name A-1
native form 6-1, 9-2
negotiate 6-2
Negotiation Glossary-5
negotiation 3-2, 4-2, 4-6, 5-1, 5-6, 10-42, A-1
Negotiation Process Glossary-4, Glossary-5
negotiation tree 5-6
Network Glossary-4, Glossary-5
network “glue” 7-6
non-terminal A-4
NULL Level Glossary-5
null level A-3, A-10
null-terminated A-7

O

Optional Encoding 6-7, 6-17 Glossary-1, Glossary-2 Glossary-3 Glossary-5
outbound 7-4
outbound data 4-6

P

page A-54
Parameter List Glossary-5
parent surface A-5, A-11
Passive Glossary-1, Glossary-5
Peripheral Glossary-1, Glossary-3, Glossary-6
physical device 3-8
plane 9-3
polled 7-7
prContentEvent E-44
preferred encodings list 5-6
prEndJob E-30
PrependBuf H-7
PrependCopyToBuf H-6
prInitialize E-24
prInRejectedEvent E-27
prJobEndEvent E-31
prJobStartEvent E-28
prReleaseSurface E-39
prRequestContent E-42
prRequestDescription E-36
prRequestSurface E-33
prSessionFromJob E-32
prShutdown E-26
prSurfaceEvent E-35
prSurfaceReleasedEvent E-41
psCancelJob E-8
psContentRequestEvent E-15
psDescriptionRequestEvent E-20
psInArrivedEvent E-4
psInGoneEvent E-5
psInitialize E-1
psJobEndEvent E-9
psRejectContentRequest E-19
psRejectSurfaceRequest E-14
psSendContent E-17
psSendDescription E-22
psSendSurface E-12
psSessionFromJob E-10
psShutdown E-3
psStartJob E-6
psSurfaceReleasedEvent E-13
psSurfaceRequestEvent E-11
push E-1
push activity E-1, E-24
Push-Receive Activity 10-43
Push-Send Activity 10-43

Q

Queue Function Reference I-1
QueueAddToFront I-5
QueueAddToRear I-4
QueueFreeAllNodes I-17
QueueInitialize I-1
QueueInsertAfter I-12
QueueInsertBefore I-14
QueueIsEmpty I-3
QueueMoveFrontToFront I-20
QueueMoveFrontToRear I-18
QueueMoveRearToFront I-19
QueueMoveRearToRear I-21
QueuePeekAtFront I-8
QueuePeekAtNext I-10
QueuePeekAtPrev I-11
QueuePeekAtRear I-9
QueueRemove I-15
QueueRemoveFront I-6
QueueRemoveRear I-7
QueueSize I-2

R

raster image 4-10
receiver E-24
Receiving Appliance Glossary-6
Reference List Glossary-6
reference list 6-7, 6-9, A-5, A-8, A-11
Reference List. Glossary-6
ReinitializeBuf H-5
Request ID 7-10
request/reply A-61, A-78
request/reply pair A-20, A-36
RLE compression A-28
RMTP 4-9
Root Attribute Block Glossary-5, Glossary-6, Glossary-7

S

security D-1
Select Glossary-6
Selection Glossary-6
Selection List Glossary-6
selection list 4-6, A-4, 6-6, 6-9,6-12, 6-13, A-2, A-8, A-9, A-13
Selection Value Glossary-6, Glossary-7, Glossary-9
self D-1
Self surface 5-2, D-1, D-11
Sending Appliance Glossary-6
sequential 9-5
serial access A-68
service layers 3-3
services 3-2
Session Glossary-6
session 4-3
session ID 7-10
split stack 3-8
srDeviceStatusEvent F-25
srInitialize F-21
srJobStatusEvent F-29
srReleaseDeviceStatus F-28
srReleaseJobStatus F-32
srSetLanguage F-23
srShutdown F-22
ssInitialize F-1
ssJobStatusReleasedEvent F-19
ssLocDeviceRequestEvent F-8
ssLocJobRequestEvent F-10
ssSendLanguage F-12
ssSetDeviceStatus F-14
ssSetJobStatus F-16
ssSetLanguages F-4
ssShutdown F-3
static image 5-6, A-14
static images 5-1
status 5-5
Status surface 5-2
Status-Receiver Activity 10-43
Status-Sender Activity 10-43
Stress Testing 10-44
Structured Value Field Glossary-6, Glossary-7, Glossary-9
subaddressing 3-8
Sub-Attribute Block Glossary-6, Glossary-7
Sub-Encoding Glossary-7
Sub-Level Glossary-3, Glossary-7
Sub-Surfac
Sub-Surface Glossary-2 Glossary-4, Glossary-7, Glossary-8
Surface 6-2, A-1, Glossary-3, Glossary-4, Glossary-7, Glossary-8, Glossary-9
Surface change 5-5
surface content 4-5, 5-1, 6-2, 6-4
Surface Description 4-5, 5-1, 6-2, 6-4, A-1, A-3, Glossary-3, Glossary-8
surface description block A-3
Surface Description Table Glossary-8
surface description table 6-27, A-4, A-5
surface handle 7-10
Surface Hierarchy 6-3, 6-8, A-5, A-11, Glossary-8
surface name 6-2
surfaces 4-1
System API H-1, I-1

T

terminal level A-4
terminal nodes A-11
Testing 10-1
text stream A-34
text, 9-3
TIAPI 8-1
TIFF A-28
top-level surface A-5
Top-Level Glossary-2
top-level encodings 9-3
Top-Level Surface 4-5, 6-8, 6-27, Glossary-4, Glossary-7, Glossary-8
Transport-Independent API 8-1
tree-like structure A-4
TrplyActiveConnect K-23
TrplyDisconnect K-32
TrplyHdr K-38
TrplyNwkStat K-11
TrplyPassiveConnect K-28
TrplyReceive K-37
TrplyReceiveDatagram K-18
TrplySend K-34
TrplySendDatagram K-15
TrplyXlateOSPAddr K-8
Type Glossary-2, Glossary-9
typedef TrplyXlateTIAddr K-10

U

undamental encoding 9-3
UniqueID J-4
unknown names B-17
UnprependBuf H-8

V

Value Glossary-6, Glossary-7, Glossary-9
Value Field Glossary-7, Glossary-9
values A-2
vArbitrary A-11, A-55
vAscii A-35
vAssociation A-14, A-76
vAttachment A-57
vColorSpace A-15

vCompression A-17
vEncoding A-14
vEndian A-14
vFile A-14, A-45
vImage A-14
vLanguag A-14
vPixelDepth A-15
vPixels A-17
vPlane A-14, A-54
vPosition A-57
vResolution A-16
vSequential A-11, A-55, A-77
vSize A-15
vText A-14, A-34
vUnicode A-35

W

Well-Known Surface 5-2, Glossary-4, Glossary-9

X

xptCall K-20
xptCancelRequest K-5
xptGetNextEvent K-39
xptHangUp K-31
xptInitialize K-3
xptListen K-26
xptNetworkEvent K-40
xptReceive K-35
xptReceiveDatagram K-17
xptRun K-6
xptSend K-33
xptSendDatagram K-13
xptShutdown K-4
xptXlateOSPAddr K-7
xptXlateTIAddr() K-9