# [MS-PAN]:
# Print System Asynchronous Notification Protocol Specification

---



## Revision Summary

| Date | Revision History | Revision Class | Comments |
|---|---|---|---|
| 10/22/2006 | 0.01 | | MCPP Milestone 1 Initial Availability |
| 01/19/2007 | 1.0 | | MCPP Milestone 1 |
| 03/02/2007 | 1.1 | | Monthly release |
| 04/03/2007 | 1.2 | | Monthly release |

| Date | Revision History | Revision Class | Comments |
|---|---|---|---|
| 05/11/2007 | 1.3 | | Monthly release |
| 06/01/2007 | 1.3.1 | Editorial | Revised and edited the technical content. |
| 07/03/2007 | 1.3.2 | Editorial | Revised and edited the technical content. |
| 07/20/2007 | 1.3.3 | Editorial | Revised and edited the technical content. |
| 08/10/2007 | 1.4 | Minor | Updated the technical content. |
| 09/28/2007 | 1.5 | Minor | Updated the technical content. |
| 10/23/2007 | 1.6 | Minor | Updated the technical content. |
| 11/30/2007 | 1.7 | Minor | Updated the technical content. |
| 01/25/2008 | 1.8 | Minor | Updated the technical content. |

# Table of Contents

# 1 Introduction

This document is a specification of the Print System Asynchronous Notification Protocol.

The Print System Asynchronous Notification Protocol is an asynchronous protocol used by clients to receive print status **notifications** from a **print server** and send any server-requested **responses** to those notifications back to the server. It is based on the **Remote Procedure Call (RPC)** protocol, as specified in [C706] and extended by the Remote Procedure Call Protocol.

A set of notifications and responses is defined together, and is referred to as a **notification type**. The RPC interfaces and methods defined by the Print System Asynchronous Notification Protocol provide a transport mechanism for arbitrary notification types.

The Print System Asynchronous Notification Protocol defines one notification type, called **AsyncUI**. The AsyncUI notification type allows for a print-server-resident **notification source** to request the display on a client of an informative alert, for a client to send user input requested by such an alert back to the server, and for the server-resident notification source to request the execution on the client of code resident on the client.

## 1.1 Glossary

The following terms are defined in [MS-GLOS]:

**Authentication**
**Authentication Level**
**Device**
**Discretionary Access Control List (DACL)**
**Generic Security Services (GSS)**
**Globally Unique Identifier (GUID)**
**HRESULT**
**Interface Definition Language (IDL)**
**Network Data Representation (NDR)**
**Opnum**
**Page Description Language (PDL)**
**Principal**
**Print Client**
**Print Job**
**Print Queue**
**Print Server**
**Print System**
**Printer Driver**
**Registration**
**Remote Procedure Call (RPC)**
**RPC Context Handle**
**RPC Dynamic Endpoint**
**Security Descriptor (SD)**
**Security Identifier (SID)**
**Security Provider**
**Unicode**
**Universal Naming Convention (UNC)**
**Universal Serial Bus (USB)**
**Universally Unique Identifier (UUID)**
**URI**
**XML**

The following terms are specific to this document:

**AsyncUI:** A **notification type** that can be used by server-resident **notification sources** to send informational alerts and user inquiries to a client component that presents them to users and to execute client-resident printer driver code.

**Bidirectional Communication Mode:** A communication mode in which a server sends **notifications** to a single client; the client replies to the **notifications**, and the server accepts that client's **response**.

**Bitmap:** A collection of structures that contain a device-independent representation of a graphical image, a logical **palette**, dimensions, and other information.

**Bitmap Resource:** A **bitmap** stored in a **resource file** that can be retrieved with a key.

**Default Resource File:** A **resource file** that is used by a client that receives **AsyncUI notifications** to look up **icons**, **bitmaps**, or **string resources**, which are referenced in **notifications** that do not explicitly name a **resource file**.<1>

**Driver-File Name:** The name of a constituent file of a printer driver previously installed on an **AsyncUI** client via **point-and-print**. A driver-file name is a directory-local name.<2>

**Icon:** A graphical image used to supplement alphanumeric text in the visual identification of an object on a computer monitor. **Icons** are typically small, relative to the size of the area on which they are displayed.

**Icon Resource:** An **icon** stored in a **resource file** that can be retrieved with a key.

**Notification:** A typed buffer of data sent by a **print server** to a client as a result of a **print server** event.

**Notification Channel:** A shareable, server-side object capable of routing **notifications** from a **print server** to appropriately registered clients.

**Notification Source:** A **print server**-resident software component, such as a **printer driver** or **port monitor** (as specified in [MS-RPRN]) that generates **notifications** conforming to a particular **notification type** or set of **notification types**, and processes any **responses** required by those **notifications**.

**Notification Type:** A set of **notification** and **response** data formats and their associated semantics. A **notification type** can be thought of as a higher-level protocol that is transported via the Print System Asynchronous Notification Protocol.

**Notification Type Identifier:** A 128-bit value that either uniquely identifies a **notification type** or is a reserved value defined for special purposes by the Print Asynchronous Notification Protocol. Although defined in **IDL** as a **GUID**, a **notification type** identifier is considered to be an opaque 128-bit value. This protocol makes no assumptions about the format of those 128 bits or about the mechanism used by the creator of a **notification type** to assure uniqueness of its **notification type identifier**.

**Palette:** An array of values, each element of which contains the definition of a color. The color elements in a **palette** are often indexed so that clients can refer to the colors, each of which can occupy 24 bits or more, by a number that requires less storage space.

**Plug-in:** An executable module that can be loaded by the **print server** to perform specific functions.

**Point-and-Print:** A method of installing network printers on a user's local computer. Point-and-print allows users to initiate a connection to a network printer, and loads any required drivers onto the client's computer. When users know which network printer they want to use, point-and-print greatly simplifies the installation process.

**Port:** A logical name that represents a connection to a **device**. A port can represent a network address (for example, a TCP/IP address) or a local connection (for example, a **Universal Serial Bus (USB)** port).

**Port Monitor:** A port monitor is a **plug-in** that is responsible for communicating with a **device** that is connected to a **port**. A port monitor may interact with the **device** locally, remotely over a network, or through some other communication channel. The data that passes through a port monitor is in a form that can be understood by the destination **device**, such as a **page description language (PDL)**.

**Position Parameter Replacement Tags:** Indicators within a string that can be replaced by parameter data during a formatting process. The indicators show which parameter of an ordered list should be used for the replacement.

**Remote Object:** An unshared, server-side object capable of representing a **registration**.

**Reply:** In **AsyncUI** names, a synonym for **response**.

**Resource File:** A file that contains one or more **icons**, **bitmaps**, or **string resources** that can be retrieved with an integer–based key and used by other software components.

**Response:** A typed buffer of data sent by the client to the server in **response** to a **notification**.

**String Resource:** A string stored in a **resource file** and that can be retrieved with a key. A string resource MAY<3> be localized into multiple languages. It is up to an **AsyncUI** client implementation to determine which language string to retrieve for a given key.<4>

**Unidirectional Communication Mode:** A communication mode in which a server sends **notifications** to a client without requesting or accepting **responses**.

**User Identity Filter:** A mechanism supported by this protocol that allows **notifications** to be directed to a particular user.

**MAY, SHOULD, MUST, SHOULD NOT, MUST NOT:** These terms (in all caps) are used as described in [RFC2119]. All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

## 1.2 References

### 1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information. Please check the archive site, http://msdn2.microsoft.com/en-us/library/E4BD6494-06AD-4aed-9823-445E921C9624, as an additional source.

[C706] The Open Group, "DCE 1.1: Remote Procedure Call", C706, August 1997, http://www.opengroup.org/public/pubs/catalog/c706.htm

[MS-DTYP] Microsoft Corporation, "Windows Data Types", January 2007.

[MS-ERREF] Microsoft Corporation, "Windows Error Codes", January 2007.

[MS-GLOS] Microsoft Corporation, "Windows Protocols Master Glossary", March 2007.

[MS-RPCE] Microsoft Corporation, "Remote Procedure Call Protocol Extensions", January 2007.

[MS-RPRN] Microsoft Corporation, "Print System Remote Protocol Specification", June 2007.

[MS-SPNG] Microsoft Corporation, "Simple and Protected Generic Security Service Application Program Interface Negotiation Mechanism (SPNEGO) Protocol Extensions", January 2007.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, http://www.ietf.org/rfc/rfc2119.txt

[RFC2616] Fielding, R., et al., "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999, http://www.ietf.org/rfc/rfc2616.txt

[RFC2781] Hoffman, P. and Yergeau, F., "UTF-16, an encoding of ISO 10646", RFC 2781, February 2000, http://www.ietf.org/rfc/rfc2781.txt

[W3C-XSD] World Wide Web Consortium, "XML Schema Part 2: Datatypes Second Edition", October 2004, http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

[XML1.0] Bray, T., Paoli, J., Sperberg-McQueen, C.M., and Maler, E., "Extensible Markup Language (XML) 1.0 (Second Edition)", W3C Recommendation, October 2000, http://www.w3.org/TR/2000/REC-xml-20001006

[XMLNS] World Wide Web Consortium, "Namespaces in XML 1.0 (Second Edition)", August 2006, http://www.w3.org/TR/REC-xml-names/

[XMLSCHEMA1/2] Thompson, H.S., Ed., Beech, D., Ed., Maloney, M., Ed., and Mendelsohn, N., Ed., "XML Schema Part 1: Structures Second Edition", W3C Recommendation, October 2004, http://www.w3.org/TR/xmlschema-1/

### 1.2.2 Informative References

[MSDN-AUTHN] Microsoft Corporation, "Authentication-Service Constants", http://msdn2.microsoft.com/en-us/library/aa373556.aspx

[MSDN-FMT] Microsoft Corporation, "FormatMessage", http://msdn2.microsoft.com/en-us/library/ms679351.aspx

## 1.3 Protocol Overview (Synopsis)

The Print System Asynchronous Notification Protocol serves two primary functions:

- A print server sending status notifications to a **print client**.
- A print server receiving the client's response to the notifications.

This protocol has two modes of operation:

- **Bidirectional communication mode**
- **Unidirectional communication mode**

In bidirectional communication mode, data can flow in two directions between a server and client. After a client registers with a server, the client requests a bidirectional **notification channel** from

the server. The client uses the channel to request predefined print status notifications from the server. When the client subsequently receives a notification, the client also uses the channel to send a response back to the server.

In bidirectional communication mode, if multiple clients open the same bidirectional notification channel and attempt to respond to the channel's initial notification, the server accepts only the first response received and continues to send further notifications only to the client whose response was accepted. Subsequent exchanges of notifications and responses on the channel take place only between the server and that client.

In the absence of a registered client prepared to receive bidirectional notifications, the server queues the notification until a client becomes available to receive it and send a notification.

The following diagram shows bidirectional communication.

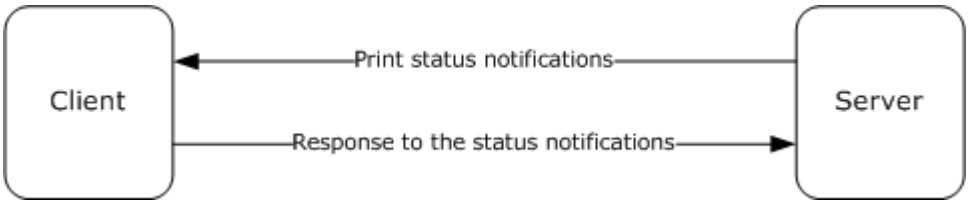


**Figure 1: Bidirectional communication**

In unidirectional communication mode, multiple clients can register for the same notifications. The server sends a given notification to all clients that have registered for it. Because unidirectional notifications do not require a response, the server can discard them in the absence of an appropriately registered client.

The following diagram shows unidirectional communication.

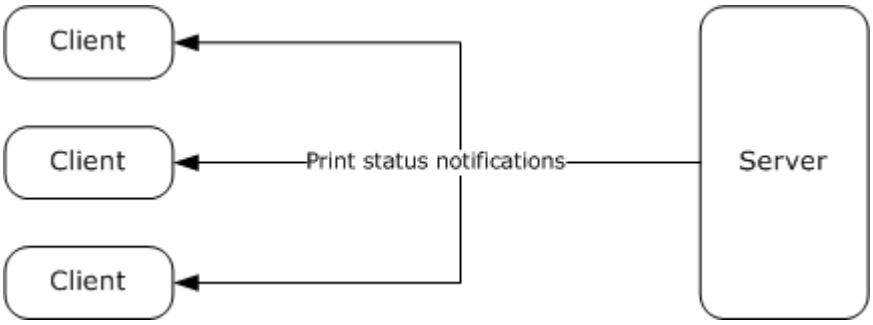


**Figure 2: Unidirectional communication**

Server-resident notification sources create, on behalf of print clients, notification channels to send notifications as printing events occur. Each channel is created to send only a given notification type in a single communication mode, unidirectional or bidirectional.

Each notification channel is created to send notifications to registered clients, irrespective of their authenticated user identity, or to send notifications to the subset of registered clients with associated authenticated user identity matching that of a specific print client. When registering for notifications, clients specify the **notification type identifier**, communication mode, and **user identity filter** for the notifications they are interested in receiving.

Unidirectional notification channels are closed only by the notification source that created the channel. Bidirectional notification channels can be closed by the client that acquired the channel or by notification source that created the channel.

The Print System Asynchronous Notification Protocol is based on the Remote Procedure Call (RPC) Protocol, and it defines the following two RPC interfaces, which are called by the client and implemented by the server:

- IRPCAsyncNotify, which is used to register and deregister clients, establish notification channels, and send data back and forth between the client and the server.
- IRPCRemoteObject, which is used to create and destroy **remote objects** that refer to printers.

This specification also defines the AsyncUI notification type, which exists to support a client component that receives and interprets notifications from server-hosted notification sources, such as **printer drivers** or **port monitors**. The AsyncUI client component can be used to display an informative message, send user input back to the notification source on the server, or trigger the execution of printer driver code on the client computer. The following diagram illustrates the relationship between a notification source and an AsyncUI client:

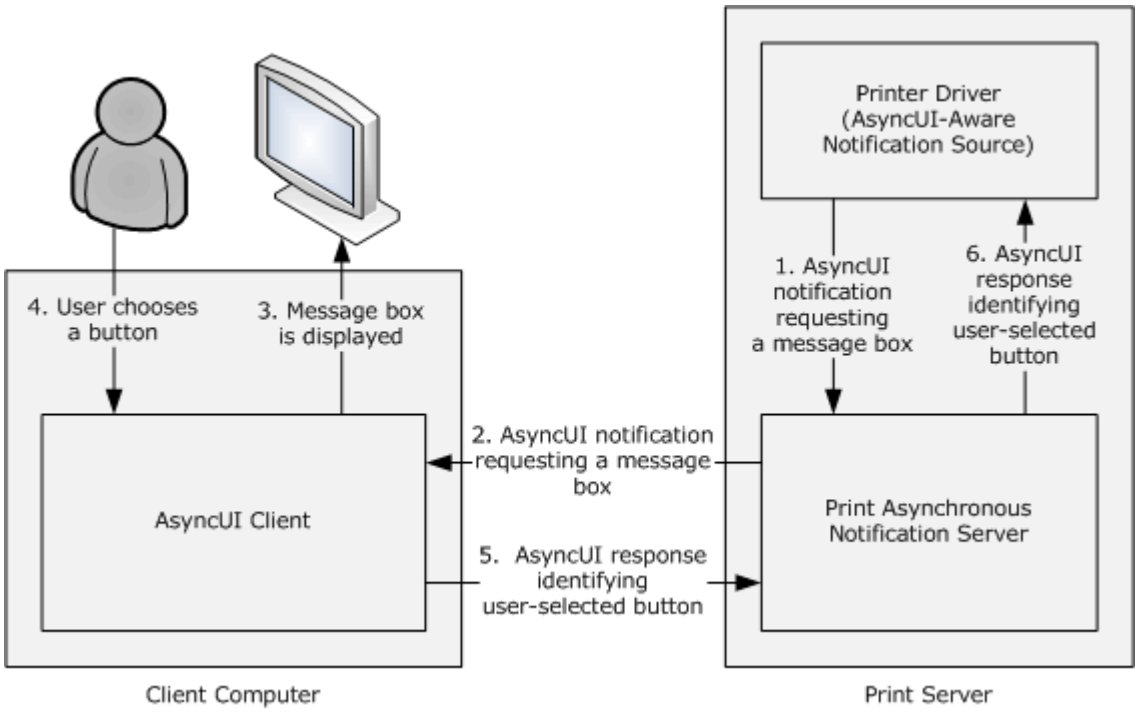


**Figure 3: Relationship between a notification source and an AsyncUI client**

## 1.4 Relationship to Other Protocols

The Print System Asynchronous Notification Protocol is dependent on the following protocols:

- Remote procedure call (RPC), as specified in [MS-RPCE]
- TCP/IP (for RPC over TCP/IP)

- Simple and Protected **Generic Security Services (GSS)** API Negotiation Mechanism (as specified in [MS-SPNG])
- Print System Remote Protocol (as specified in [MS-RPRN])

No protocols are dependent on the Print System Asynchronous Notification Protocol.

## 1.5 Prerequisites/Preconditions

This protocol defines RPC interfaces and therefore has the prerequisites common to RPC interfaces, as specified in [MS-RPCE] section 1.5.

This document assumes that the protocol client has obtained the name of a server that supports the Print System Asynchronous Notification Protocol before this protocol is invoked.

This document assumes that the server generating AsyncUI notifications, and the clients receiving them, agree on **resource files**, resource keys within those files, and positional parameters within **string resources** that are referenced in those notifications.

## 1.6 Applicability Statement

The Print System Asynchronous Notification Protocol is applicable only for printing operations between a machine functioning as a client and a machine functioning as a print server. The protocol is intended for the communication of notifications and responses between notification sources operating on a print server, and client applications.

The protocol can be used in a broad set of scenarios ranging from home-use, where one computer makes its printer available for use by other computers, to enterprise-use, where a print server provides printing services for many computers.

The protocol is not applicable outside client/server printing and monitoring print operations.

## 1.7 Versioning and Capability Negotiation

This document covers versioning issues in the following areas:

**Supported Transports**: The Print System Asynchronous Notification Protocol uses RPC over TCP only, as specified in section 2.1.

**Protocol Versions**: There is only one version of this protocol. It has a built-in versioning and extensibility feature that can be used to send and receive new data formats by defining new notification types and creating associated notification type identifiers, as specified in section 2.2.1.

**Security and Authentication Methods**: This protocol uses Simple and Protected Generic Security Service Application Program Interface Negotiation Mechanism (SPNEGO) Protocol Extensions and RPC packet **authentication level** for security and **authentication**, as specified in section 2.1.

**Localization**: The AsyncUI notification types pass string resource keys in various message data formats. Localization considerations for the associated string resources are specified in section 1.1.

**Capability Negotiation**: There is no capability negotiation mechanism built into the protocol itself; however, a vendor can define a new notification type identifier and associate it with a set of notification and response data formats and sequencing rules, as specified in section 2.2.1.

## 1.8 Vendor-Extensible Fields

This protocol uses **HRESULT** method return values as specified in [MS-ERREF]. Vendors are free to choose their own values for this field, but the C bit (0x20000000) MUST be set, indicating it is a customer code.<5>

Unless otherwise specified in this document, a client of this protocol MUST NOT interpret returned error codes. The client MUST simply return error codes to the invoking application without taking any protocol action.

The set of notification types used by the Print System Asynchronous Notification Protocol is extensible. New notification types can be defined and associated with new notification type identifier values. This mechanism, as specified in section 2.2.1, enables future versioning and extensibility.

## 1.9 Standards Assignments

| Parameter | Value | Reference |
|---|---|---|
| RPC **UUID** for IRPCAsyncNotify | 0b6edbfa-4a24-4fc6-8a23-942b1eca65d1 | As specified in [C706], Appendix A; for more information, see section 3.1 |
| RPC UUID for IRPCRemoteObject interface | ae33069b-a2a8-46ee-a235-ddfd339be281 | As specified in [C706] Appendix A; for more information, see section 3.3 |

# 2 Messages

The following sections specify how Print System Asynchronous Notification Protocol messages are encapsulated on the wire, and common Print System Asynchronous Notification Protocol data types.

## 2.1 Transport

The Print System Asynchronous Notification Protocol MUST use the transport RPC over TCP/IP, as specified in [MS-RPCE] section 2.1.1.1.

This protocol MUST use **RPC dynamic endpoint**, as specified in [C706], chapter 6.

This protocol MUST use the UUIDs as specified in section 1.9.

A server of this protocol MUST use:

- A **security provider** that supports SPNEGO Protocol Extensions, as specified in [MS-RPCE] section 3 and [MS-SPNG].
- The default server **principal** name for the security provider, which is the authentication-service constant **RPC_C_AUTHN_GSS_NEGOTIATE**. For information concerning Windows authentication-service constants, see [MSDN-AUTHN].

A client of this protocol MUST use:

- A security provider that supports SPNEGO Protocol Extensions, as specified in [MS-RPCE] section 3 and [MS-SPNG].
- A principal name constructed by appending the name of the print server to the string "host/".
- Packet authentication level, as specified in [MS-RPCE] section 3.

## 2.2 Common Data Types

This protocol MUST indicate to the RPC runtime that it is to support both the **NDR** and NDR64 transfer syntaxes, and provide a negotiation mechanism for determining which transfer syntax will be used, as specified in [MS-RPCE] section 3.

In addition to RPC base types and definitions specified in [C706] section 4.2.9 and [MS-RPCE] section 2.2.1, additional data types are defined in the following sections.

### 2.2.1 PrintAsyncNotificationType

The **PrintAsyncNotificationType** data type supports the definition of unique functional categories of notifications for the Print System Asynchronous Notification Protocol. This type is used for matching notifications from the server to appropriate clients.

This type is declared as follows:

```
typedef GUID PrintAsyncNotificationType;
```

**PrintAsyncNotificationType** MUST be a notification type identifier. When a new notification type is defined, its creator MAY<6> use any mechanism to assure uniqueness of its notification type identifier, including the algorithm specified in [C706] Appendix A.

This protocol defines a single, reserved notification type identifier value. This value is not associated with any specific set of notification and response data formats, but rather has special meaning in the definition of this protocol.

| Name/value | Description |
|---|---|
| **NOTIFICATION_RELEASE**<br>ba9a5027-a70e-4ae7-9b7d-eb3e06ad4157 | This value indicates that a client or server is not accepting further communication.<br>For more information, see sections 3.1.4.4, 3.1.4.5, and 3.1.4.6. |

This protocol also defines the notification and response data formats for the AsyncUI notification type. Associated with the AsyncUI notification type is its notification type identifier.

| Name/value | Description |
|---|---|
| **AsyncPrintNotificationType_AsyncUI**<br>f6853f92-eb31-4e23-b6e7-fd69056153f0 | This value indicates that the notification data byte arrays contain AsyncUI data formats.<br>For more information, see sections 2.3, 3.5, and 3.6. |

### 2.2.2 PrintAsyncNotifyUserFilter

The **PrintAsyncNotifyUserFilter** enumeration is used by clients when they register to receive notifications from server-resident notification sources. The following types of notifications can be requested:

- Notifications intended specifically for a particular client's user identity;
- Notifications intended for all registered client user identities.

A server SHOULD consider the security and privacy context prior to letting users monitor and receive notifications for all user identities.<7>

```
typedef [v1_enum] enum
{
  kPerUser = 0,
  kAllUsers = 1
} PrintAsyncNotifyUserFilter;
```

**kPerUser:** Indicates that the client is requesting notifications that are intended specifically for its own user identity and notifications that are intended for all registered user identities.

**kAllUsers:** Indicates that the client is requesting every notification, whether intended for a specific user identity or all registered user identities.

### 2.2.3 PrintAsyncNotifyConversationStyle

The **PrintAsyncNotifyConversationStyle** enumeration MUST specify the communication mode expected between the sender and a registered client.

```
typedef [v1_enum] enum
{
  kBiDirectional = 0x00000000,
  kUniDirectional = 0x00000001
} PrintAsyncNotifyConversationStyle;
```

**kBiDirectional:** Bidirectional communication mode is specified. The sender expects the client to send responses to notifications.

**kUniDirectional:** Unidirectional communication mode is specified. The sender does not expect the client to respond to notifications.

### 2.2.4 PRPCREMOTEOBJECT

The **PRPCREMOTEOBJECT** data type defines an **RPC context handle**, which corresponds to the server remote object representing a client **registration**. A client MUST call **IRPCRemoteObject_Create** to create a **PRPCREMOTEOBJECT** handle, and **IRPCRemoteObject_Delete** to delete a **PRPCREMOTEOBJECT** handle. For more information, see section 3.3.4.

This type is declared as follows:

```
typedef [context_handle] void* PRPCREMOTEOBJECT;
```

### 2.2.5 PNOTIFYOBJECT

The **PNOTIFYOBJECT** data type defines an RPC context handle, which corresponds to the server object representing a notification channel. This handle is used in bidirectional communication mode only.

This type is declared as follows:

```
typedef [context_handle] void* PNOTIFYOBJECT;
```

## 2.3 AsyncUI XML Notification and Response Formats

This section specifies the data formats for notifications and responses associated with the notification type identifier value **AsyncPrintNotificationType_AsyncUI**, as specified in section 2.2.1. The data formats are specified by using a combination of prose and **XML** schema syntax, as specified in [W3C-XSD].

The XML schema fragments contained in this section are drawn from two separate XML schema documents, one for AsyncUI notifications and one for AsyncUI responses. Both schema documents specify a value of "qualified" for the **elementFormDefault** attribute of the root "schema" element.

The XML schema document for AsyncUI notifications MUST specify a **targetNamespace** attribute value of "http://schemas.microsoft.com/2003/print/asyncui/v1/request", and also MUST use that **URI** as the schema document's default namespace (as specified in [XMLNS] sections 2.1 and 3.0).

The XML schema document for AsyncUI responses MUST specify a **targetNamespace** attribute value of "http://schemas.microsoft.com/2003/print/asyncui/v1/response", and also MUST use that URI as the schema document's default namespace.

Server-resident notification sources such as printer drivers can use the AsyncUI notification type to display printing-related, interactive UIs on client systems.

The XML data contained within AsyncUI notifications and responses MUST obey the syntax of well-formed XML 1.0 documents, as specified in [XML1.0] section 2.1 . Furthermore, those documents MUST be encoded in UTF-16LE format, as specified in [RFC2781] section 4.2.

Note that XML 1.0, as specified in [XML1.0], restricts the set of legal characters that can be used. Values that cannot be expressed in XML MUST NOT be represented in an XML document contained within an AsyncUI notification or response. Furthermore, some legal characters, such as "<" and "&", require some form of escaping when encoded in XML. Implementations MUST use an XML-defined mechanism, such as CDATA (as specified in [[XML1.0] sections 2.4 and 2.7) or numeric character references (as specified in [XML1.0] section 4.1), to encode such characters within XML documents.

### 2.3.1 Common AsyncUI Elements

The following sections define XML elements common to some of the AsyncUI notification data formats.

#### 2.3.1.1 asyncPrintUIRequest Element

The **asyncPrintUIRequest** XML element MUST be the root element of XML documents used in the **AsyncUIBalloon**, **AsyncUIMessageBox**, **AsyncUICustomUI**, and **AsyncUICustomData** server-to-client notification formats, as specified in sections 2.3.2, 2.3.3, 2.3.5, and 2.3.7, respectively.

The document markup MUST be schema-valid according to the following XML schema, which refers to additional schema fragments specified in section 2.3; Schema-Validity Assessment of the document's root element MUST result in a value of "valid" for the **[validity]** property, as defined in [XMLSCHEMA1/2] section 3.3.5.

```
<xs:element name="asyncPrintUIRequest">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="v1">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="requestOpen">
              <xs:complexType>
                <xs:choice>
                  <xs:element
                    ref="balloonUI"
                   />
                  <xs:element
                    ref="messageBoxUI"
                   />
                  <xs:element
                    ref="customUI"
```

```
                    />
                  <xs:element
                    ref="customData"
                    />
                </xs:choice>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Child Elements

| Element | Type | Description |
|---|---|---|
| v1 | N/A | A required element within an **asyncPrintUIRequest** element that MUST contain exactly one requestOpen element. |
| requestOpen | N/A | A required element that MUST contain exactly one **balloonUI**, **messageBoxUI**, **customUI**, or **customData** element. |
| **balloonUI** | **balloonUI** | See section 2.3.2.2. |
| **messageBoxUI** | **messageBoxUI** | See section 2.3.3.4. |
| **customUI** | **customUI** | See section 2.3.5.1. |
| **customData** | **customData** | See section 2.3.7.1. |

### 2.3.1.2 asyncPrintUIResponse Element

The **asyncPrintUIResponse** XML element MUST be the root element of the XML documents used in the AsyncUIMessageBoxReply and AsyncUICustomUIReply client-to-server response formats, as specified in sections 2.3.4 and 2.3.6, respectively.

The document markup MUST be schema-valid according to the following XML schema, which refers to additional schema fragments specified in section 2.3; Schema-Validity Assessment of the document's root element MUST result in a value of "valid" for the **[validity]** property, as defined in [XMLSCHEMA1/2] section 3.3.5.

```
<xs:element name="asyncPrintUIResponse">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="v1">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="requestClose">
              <xs:complexType>
                <xs:choice>
                  <xs:element
                    ref="CustomUI"
```

```
                    />
                   <xs:element
                     ref="messageBoxUI"
                    />
                 </xs:choice>
               </xs:complexType>
             </xs:element>
           </xs:sequence>
         </xs:complexType>
       </xs:element>
     </xs:sequence>
   </xs:complexType>
 </xs:element>
```

Child Elements

| Element | Type | Description |
|---|---|---|
| v1 | N/A | A required element within an **asyncPrintUIResponse** element that MUST contain exactly one requestClose element. |
| requestClose | N/A | A required element that MUST contain exactly one **messageBoxUI** or **CustomUI** element. |
| **CustomUI** | **CustomUI** | See section 2.3.6.1. |
| **messageBoxUI** | **messageBoxUI** | See section 2.3.3.4. |

### 2.3.1.3 title Element

The **title** XML element MUST specify a string, using attributes or nested text, optionally combined with nested **parameter** elements, that SHOULD be used on the client as the displayable title of a printer event.<8>

```
<xs:element name="title">
  <xs:complexType
    mixed="true"
  >
    <xs:sequence>
      <xs:element
        minOccurs="0"
        maxOccurs="unbounded"
        ref="parameter"
       />
    </xs:sequence>
    <xs:attribute name="stringID"
      type="xs:integer"
      use="optional"
     />
    <xs:attribute name="resourceDll"
      type="xs:string"
      use="optional"
     />
  </xs:complexType>
```

```
</xs:element>
```

Child Elements

| Element | Type | Description |
|---|---|---|
| **parameter** | **parameter** | See section 2.3.1.5. |

Attributes

| Name | Type | Description |
|---|---|---|
| stringID | xs:integer | The value of this optional attribute, if present, MUST be the key of a string resource in the resource file specified by the **resourceDll** attribute. If the **resourceDll** attribute is not specified, **stringID** MUST be the key of a string resource in a default resource file.<br>The string MAY<9> contain position parameter replacement tags. |
| resourceDll | xs:string | The value of this optional attribute, if present, MUST be the driver-file name of a resource file on the client system, which MUST contain the string resources used in this message.<br>If no value is specified, a default resource file MUST be used. |

If the **stringID** attribute is not specified, the text content in the **title** element MUST be present and MUST be used by the client as the string to display. The nested text MAY contain position parameter replacement tags. If so, the client MUST replace them with the strings constructed from the sequence of **parameter** elements.

If the **stringID** attribute is specified, the **title** element MUST NOT contain nested text, and the client MUST treat the presence of such text as an error.

Nested text MUST NOT follow any **parameter** element.

### 2.3.1.4 body Element

The **body** XML element MUST specify a string, using attributes or nested text, optionally combined with nested **parameter** elements, that SHOULD represent the displayable description of a printer event.

A single notification MAY<10> contain multiple **body** elements. The client MUST concatenate the text resulting from the processing of each successive **body** element to determine the complete displayable description. The client SHOULD interpose a single space character between each pair of concatenated strings.

```
<xs:element name="body">
  <xs:complexType
    mixed="true"
  >
    <xs:sequence>
      <xs:element
        minOccurs="0"
        maxOccurs="unbounded"
        ref="parameter"
```

```
      />
    </xs:sequence>
    <xs:attribute name="stringID"
      type="xs:integer"
      use="optional"
     />
    <xs:attribute name="resourceDll"
      type="xs:string"
      use="optional"
     />
  </xs:complexType>
</xs:element>
```

Child Elements

| Element | Type | Description |
|---|---|---|
| **parameter** | **parameter** | See section 2.3.1.5. |

Attributes

| Name | Type | Description |
|---|---|---|
| stringID | xs:integer | The value of this optional attribute, if present, MUST be the key of a string resource in the resource file specified by the resourceDll attribute. If the resourceDll attribute is not specified, stringID MUST be the key of a string resource or in a default resource file.<br>The string resource MAY<11> contain position parameter replacement tags. If so, the client MUST replace them with the strings constructed from the sequence of parameter elements. |
| resourceDll | xs:string | The value of this optional attribute, if present, MUST be the driver-file name of a resource file on the client system, which MUST contain the string resources used in this message. If no value is specified, a default resource file MUST be used. |

If the **stringID** attribute is not specified, the text content in the **body** element MUST be present and MUST be used by the client as the string to display. The nested text MAY contain position parameter replacement tags. If so, the client MUST replace them with the strings constructed from the sequence of **parameter** elements.

If the **stringID** attribute is specified, the **body** element MUST NOT contain nested text, and the client MUST treat the presence of such text as an error.

Nested text MUST NOT follow any **parameter** element.

### 2.3.1.5 parameter Element

The **parameter** XML element MUST specify a string, using attributes or nested text, that SHOULD provide additional information when formatting a string specified by a **body** or **title** element.

```
<xs:element name="parameter">
  <xs:complexType>
```

```
      <xs:simpleContent>
        <xs:extension
          base="xs:string"
        >
          <xs:attribute name="type"
            type="xs:string"
            use="optional"
           />
          <xs:attribute name="stringID"
            type="xs:integer"
            use="optional"
           />
          <xs:attribute name="resourceDll"
            type="xs:string"
            use="optional"
           />
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
```

Attributes

| Name | Type | Description |
|---|---|---|
| type | xs:string | The value of this optional attribute, if present, MUST be the string "PrinterName". If a value of "PrinterName" is specified, the client MUST ignore the **stringID** and **resourceDll** attributes. For string-resource-formatting purposes, the client SHOULD use a recognizable name for the printer that triggered the notification. |
| stringID | xs:integer | The value of this optional attribute, if present, MUST be the key of a string resource in the resource file specified by the **resourceDll** attribute. If the **resourceDll** attribute is not specified, **stringID** MUST be the key of a string resource in a default resource file. |
| resourceDll | xs:string | The value of this optional attribute, if present, MUST be the driver-file name of a resource file on the client system, which MUST contain the string resources used in this message. If neither a **type** attribute nor a **resourceDll** attribute is specified, a default resource file MUST be used. |

If neither the **type** nor the **stringID** attribute is specified, the text content in the **parameter** element MUST be present and MUST be used by the client when processing position parameter replacement tags within a string specified by a **title** or **body** element.

If either the **type** or the **stringID** attribute is specified, the **parameter** element SHOULD NOT contain nested text, and the client SHOULD<12> treat the presence of such text as an error.

### 2.3.2 AsyncUIBalloon

AsyncUIBalloon is a string that MUST contain a well-formed XML document, as specified in [XML1.0] section 2.1. The root element of the document MUST be the asyncPrintUIRequest element, as specified by the schema in section 2.3.1.1. A **balloonUI** element MUST be nested within the **AsyncPrintUIRequest** markup, at the point where it is referenced in the XML schema.

AsyncUIBalloon SHOULD be used by a printer driver on a server to deliver to a client UI the details of an event or change in device status.

### 2.3.2.1 action Element

The **action** XML element MUST identify a specific entry point in a specific file on the client computer that MUST contain executable code.

```
<xs:element name="action">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension
        base="xs:string"
      >
        <xs:attribute name="dll"
          type="xs:string"
          use="required"
         />
        <xs:attribute name="entrypoint"
          type="xs:string"
          use="required"
         />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

Attributes

| Name | Type | Description |
|---|---|---|
| dll | xs:string | The value of this attribute MUST be a string, which MUST contain the driver-file name of a file on the client system that contains executable code. When calling a method that is identified by an **entrypoint** attribute, clients SHOULD restrict the set of acceptable **dll** attribute values and active permissions, to minimize the risk of executing arbitrary client-resident code.<13> |
| entrypoint | xs:string | The value of this attribute MUST be a string, which MUST specify a public method in the file designated by the **dll** attribute.<14> The **action** element SHOULD<15>contain text data to pass to the **entrypoint** method as a parameter. |

### 2.3.2.2 balloonUI Element

The **balloonUI** XML element MUST contain information on a printer event that can be displayed on the client computer. The notification MUST contain references to string resources that pertain to a printer event.

A **balloonUI** element can optionally call for the execution of code on the client system, by means of a nested **action** element.

```
<xs:element name="balloonUI">
  <xs:complexType>
    <xs:sequence>
```

```
        <xs:element
          ref="title"
         />
        <xs:element
          maxOccurs="unbounded"
          ref="body"
         />
        <xs:element
          minOccurs="0"
          ref="action"
         />
      </xs:sequence>
      <xs:attribute name="iconID"
        type="xs:integer"
        use="optional"
       />
      <xs:attribute name="resourceDll"
        type="xs:string"
        use="optional"
       />
    </xs:complexType>
  </xs:element>
```

Child Elements

| Element | Type | Description |
|---|---|---|
| **title** | **title** | See section 2.3.1.3. |
| **body** | **body** | See section 2.3.1.4. |
| **action** | **action** | See section 2.3.2.1 |

Attributes

| Name | Type | Description |
|---|---|---|
| iconID | xs:integer | The value of this optional attribute, if present, MUST be the key of an icon resource in the resource file specified by the **resourceDll** attribute. If an **iconID** is provided, a **resourceDll** MUST also be specified.<16> |
| resourceDll | xs:string | The value of this optional attribute, if present, MUST be the driver-file name of a resource file on the client system that MUST contain the icon resource specified by the **iconID** attribute. If this attribute is not specified, a generic icon SHOULD be used.<17> |

### 2.3.3 AsyncUIMessageBox

AsyncUIMessageBox is a string that MUST contain a well-formed XML document, as specified in [XML1.0] section 2.1.

The root element of the document MUST be the **asyncPrintUIRequest** element, as specified by the schema in section 2.3.1.1. A **messageBoxUI** element MUST be nested within the **AsyncPrintUIRequest** markup, at the point where it is referenced in the XML schema.

AsyncUIMessageBox SHOULD be used by a printer driver on a server to deliver to a client UI the details of an event or change in device status, and to request input from the user to guide its handling of the event or change in status.

### 2.3.3.1 button Element

The **button** XML element MUST contain information on a button control in the UI that SHOULD be displayed on the client computer.

```
<xs:element name="button">
  <xs:complexType>
    <xs:sequence />
    <xs:attribute name="stringID"
      type="xs:integer"
      use="optional"
     />
    <xs:attribute name="resourceDll"
      type="xs:string"
      use="optional"
     />
    <xs:attribute name="buttonID"
      use="required"
    >
      <xs:simpleType>
        <xs:restriction
          base="xs:string"
        >
          <xs:pattern
            value="IDOK|IDCANCEL|\s*(\-|\+)?[0-9]+\s*"
           />
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
```

Attributes

| Name | Type | Description |
|---|---|---|
| stringID | xs:integer | This attribute MUST be present if the value of the **buttonID** attribute is neither "IDOK" nor "IDCANCEL". If present, the value of this attribute MUST be the key of a string resource in the resource file specified by the **resourceDll** attribute. If the **resourceDll** attribute is not specified, **stringID** MUST be the key of a string resource in a default resource file. |
| resourceDll | xs:string | The value of this optional attribute, if present, MUST be the driver-file name of a resource file on the client system, which MUST contain the string resources used in this message. If no value is specified, a default resource file MUST be used. |
| buttonID | | The value of this attribute MAY<18> be one of the following case-sensitive string values; if so, the **stringID** and **resourceDll** values MUST be ignored. If the value is not one of these strings, then it MUST be a string representation of an integer. |

| Name | Type | Description |
|---|---|---|
| | | "IDOK" The button SHOULD<19> correspond to OK button behavior in the dialog.<br>"IDCANCEL" The button SHOULD<20> correspond to Cancel button behavior in the dialog. |

### 2.3.3.2 buttons Element

The **buttons** XML element MUST contain a collection of **button** elements. The client SHOULD display these buttons to present the user with a choice of actions.

```
<xs:element name="buttons">
  <xs:complexType>
    <xs:sequence>
      <xs:element
        maxOccurs="5"
        ref="button"
       />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Child Elements

| Element | Type | Description |
|---|---|---|
| **button** | **button** | See section 2.3.3.1. |

### 2.3.3.3 bitmap Element

The **bitmap** XML element, if present, MUST specify a **bitmap** that SHOULD be displayed on the client computer, to describe a printer event.

This element can be used to add an informational graphic to the UI for printer events. For example, in the case of a paper jam, a schematic could be displayed that indicates where the paper jam occurred.

```
<xs:element name="bitmap">
  <xs:complexType>
    <xs:sequence />
    <xs:attribute name="bitmapID"
      type="xs:integer"
      use="required"
     />
    <xs:attribute name="resourceDll"
      type="xs:string"
      use="optional"
     />
  </xs:complexType>
</xs:element>
```

Attributes

| Name | Type | Description |
|---|---|---|
| bitmapID | xs:integer | The value of this attribute MUST be the key of a bitmap resource in the resource file specified by the **resourceDll** attribute. If no **resourceDll** value is specified, **bitmapID** MUST be the key of a bitmap resource in a default resource file. |
| resourceDll | xs:string | The value of this optional attribute MUST be the driver-file name of a resource file on the client system that MUST contain the bitmap resource used in this message. |

### 2.3.3.4 messageBoxUI Element

The **messageBoxUI** XML element MUST specify elements that compose a message box for display in a client UI.<21>

```
<xs:element name="messageBoxUI">
  <xs:complexType>
    <xs:sequence>
      <xs:element
        ref="title"
       />
      <xs:element
        minOccurs="0"
        ref="bitmap"
       />
      <xs:element
        maxOccurs="unbounded"
        ref="body"
       />
      <xs:element
        ref="buttons"
       />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Child Elements

| Element | Type | Description |
|---|---|---|
| **title** | **title** | See section 2.3.1.3. |
| **bitmap** | **bitmap** | See section 2.3.3.3 |
| **body** | **body** | See section 2.3.1.4. |
| **buttons** | **buttons** | See section 2.3.3.2 |

### 2.3.4 AsyncUIMessageBoxUIReply

AsyncUIMessageBoxUIReply is a string that MUST contain a well-formed XML document, as specified in [XML1.0] section 2.1.

The root element of the document MUST be an **asyncPrintUIResponse** element. A **messageBoxUI** element MUST be nested within the **asyncPrintUIResponse** markup, at the point where it is referenced in the XML schema.

AsyncUIMessageBoxUIReply MUST carry the response from a client to an AsyncUIMessageBoxUIReply notification.

#### 2.3.4.1 buttonID Element

The **buttonID** XML element MUST contain an integer, which specifies the **button** element from the AsyncUIMessageBox string that was selected by the user.

If the selected **button** element has a **buttonID** attribute value of "IDOK", the **buttonID** element MUST contain the value 1. If the selected **button** element has a **buttonID** attribute value of "IDCANCEL", the **buttonID** element MUST contain the value 2. If the selected **button** element holds an integer value, the **buttonID** element MUST contain that value.

```
<xs:element name="buttonID"
  type="xs:integer"
 />
```

#### 2.3.4.2 messageBoxUI Element

The **messageBoxUI** XML element MUST contain a **buttonID** element that MUST contain information identifying the button that was selected by the user.

```
<xs:element name="messageBoxUI">
  <xs:complexType>
    <xs:sequence>
      <xs:element
        ref="buttonID"
       />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Child Elements

| Element | Type | Description |
|---|---|---|
| **buttonID** | **buttonID** | See section 2.3.4.1 |

### 2.3.5 AsyncUICustomUI

AsyncUICustomUI is a string that MUST contain a well-formed XML document, as specified in [XML1.0] section 2.1.

The root element of the document MUST be an **asyncPrintUIRequest** element, as specified by the schema in section 2.3.1.1. A **customUI** element MUST be nested within the **asyncPrintUIRequest** markup, at the point where it is referenced in the XML schema.

AsyncUICustomUI (or the similar AsyncUICustomData) SHOULD be used by a printer driver on a server when it requires client-side handling of an event or change in device status that cannot be expressed by using an AsyncUIMessageBox or AsyncUIBalloon notification. The AsyncUICustomUI notification calls for the execution of client-resident code that is associated with the server-resident printer driver.

### 2.3.5.1 customUI Element

The **customUI** XML element MUST specify a method to call and the data to supply as an argument to that method.

```
<xs:element name="customUI">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension
        base="xs:string"
      >
        <xs:attribute name="dll"
          type="xs:string"
          use="required"
         />
        <xs:attribute name="entrypoint"
          type="xs:string"
          use="required"
         />
        <xs:attribute name="bidi"
          use="required"
        >
          <xs:simpleType>
            <xs:restriction
              base="xs:string"
            >
              <xs:enumeration
                value="true"
               />
              <xs:enumeration
                value="false"
               />
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

Attributes

| Name | Type | Description |
|---|---|---|
| dll | xs:string | The value of this attribute MUST be a string, which MUST contain the driver-file name of a file on the client system that contains executable code. When calling a method that is identified by an **entrypoint** attribute, clients SHOULD<22> restrict the set of acceptable **dll** attribute values and active permissions, to minimize the risk of executing arbitrary client-resident code. |
| entrypoint | xs:string | The value of this attribute MUST be a string, which MUST specify a public method<23> in the file designated by the **dll** attribute. The **customUI** element SHOULD contain text data to pass<24> to the **entrypoint** method as a parameter. If the **bidi** attribute value is "true", the **entrypoint** method MUST return a value to be encoded as text<25> in the AsyncUICustomUIReply response to this notification. |
| bidi | enumeration | The value of this attribute MUST specify whether the client is expected to send a response to the server. The case-sensitive string MUST be one of the following values.<br>A value of "true" indicates that the notification containing the **customUI** element MUST have been sent over a bidirectional notification channel, and the client MUST send a response back to the print server. The method specified by **entrypoint** MUST return a string, which MUST be returned to the server in an AsyncUICustomUIReply response string (for more information, see section 2.3.6).<br>A value of "false" indicates that the notification containing the **customUI** element MUST NOT have been sent over a bidirectional notification channel, and the client MUST NOT send a response back to the print server.<br>**Value** / **Description**<br>true<br>false |

### 2.3.6 AsyncUICustomUIReply

AsyncUICustomUIReply is a string containing a well-formed XML document, as specified in [XML1.0] section 2.1.

The root element of the document MUST be the **asyncPrintUIResponse** element, as specified by the schema in section 2.3.1.2. A **CustomUI** element MUST be nested within the **asyncPrintUIResponse** markup, at the point where it is referenced in the XML schema.

AsyncUICustomUIReply MUST carry the response from a client to an AsyncUICustomUI or AsyncUICustomData notification.

#### 2.3.6.1 CustomUI Element

The **CustomUI** XML element MUST contain text that encodes the value returned by the call to the **entrypoint** method, which is identified in the **customUI** element of an AsyncUICustomUI notification or in the **customData** element of an AsyncUICustomData notification.

```
<xs:element name="CustomUI"
```

```
    type="xs:string"
  />
```

### 2.3.7 AsyncUICustomData

AsyncUICustomData MUST be a null-terminated string containing a well-formed XML document, as specified in [XML1.0] section 2.1, followed by binary data.

The root element of the document MUST be the **asyncPrintUIRequest** element, as specified by the schema in section 2.3.1.1. A **customData** element MUST be nested within the **asyncPrintUIRequest** markup, at the point where it is referenced in the XML schema.

The entry point specified in the **customData** element MUST be called, passing the binary data as an argument.

AsyncUICustomData can be used in any case where a printer driver on a server might choose to use AsyncUICustomUI. But because AsyncUICustomData encodes the data passed to the specified entry point in binary form, the AsyncUICustomData notification can transport argument values that cannot be expressed in legal XML within an AsyncUICustomUI notification.

#### 2.3.7.1 customData Element

The **customData** XML element MUST specify a method entry point to call.

```
<xs:element name="customData">
  <xs:complexType>
    <xs:sequence />
    <xs:attribute name="dll"
      type="xs:string"
      use="required"
     />
    <xs:attribute name="entrypoint"
      type="xs:string"
      use="required"
     />
    <xs:attribute name="bidi"
      use="required"
    >
      <xs:simpleType>
        <xs:restriction
          base="xs:string"
        >
          <xs:enumeration
            value="true"
           />
          <xs:enumeration
            value="false"
           />
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
```

Attributes

| Name | Type | Description |
|---|---|---|
| dll | xs:string | The value of this attribute MUST be a string that MUST specify the name of a file on the client system that contains executable code. When calling a method that is identified by an **entrypoint** attribute, clients SHOULD<26> restrict the set of acceptable **dll** attribute values and active permissions, to minimize the risk of executing arbitrary client-resident code. |
| entrypoint | xs:string | The value of this attribute MUST be a string that MUST specify the entry point of a public method<27> in the file designated by the **dll** attribute. The AsyncUICustomData notification SHOULD contain binary data following the null-terminated XML document. This binary data SHOULD be passed to the **entrypoint** method as a parameter. If the **bidi** attribute value is "true", the **entrypoint** method MUST return a value to be encoded as text<28> in the AsyncUICustomUIReply response to this notification. |
| bidi | enumeration | The value of this attribute MUST specify whether the client is expected to send a response to the server. The case-sensitive string MUST be one of the following values.<br>"true" Indicates that the notification containing the **customData** element MUST have been sent over a bidirectional notification channel, and the client MUST send a response back to the print server. The method specified by **entrypoint** MUST return a string, which MUST be returned to the server in an AsyncUICustomUIReply response string (for more information, see section 2.3.6).<br>"false" Indicates that the notification containing the **customData** element MUST NOT have been sent over a bidirectional notification channel, and the client MUST NOT send a response back to the print server. |

| Value | Description |
|---|---|
| true | |
| false | |

# 3 Protocol Details

The following sections specify details of the Print System Asynchronous Notification Protocol, including abstract data models, interface method syntax, and message processing rules.

## 3.1 Asynchronous Notification Server Details

The asynchronous notification server interface is identified by UUID 0b6edbfa-4a24-4fc6-8a23-942b1eca65d1.

Unidirectional message passing mode is illustrated by the following server state diagram.

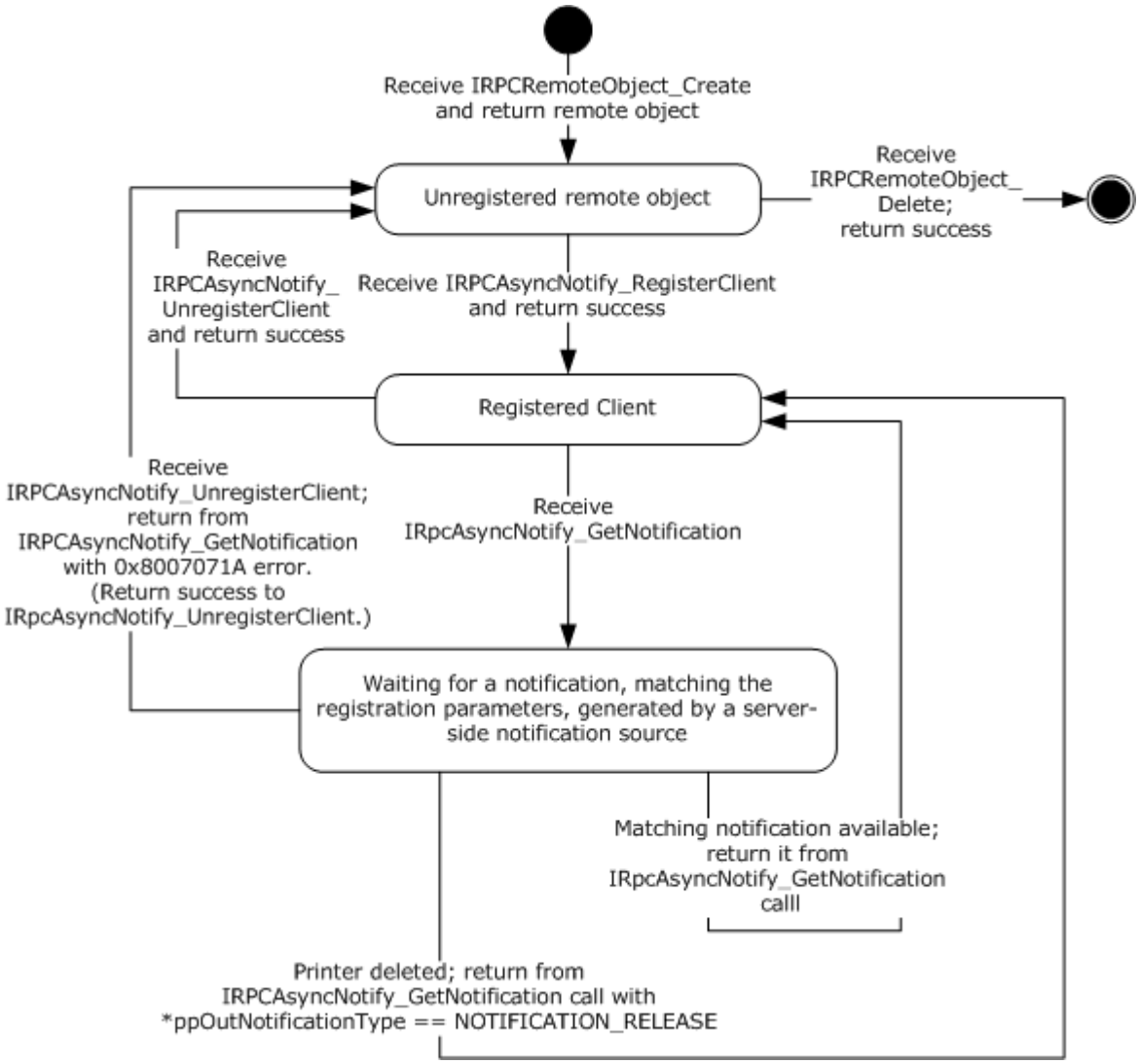


**Figure 4: Unidirectional message passing mode**

Bidirectional message passing mode is illustrated by the following two server state diagrams. The first diagram illustrates remote object creation and deletion, client registration, and the opening of

notification channels. The second diagram details the processing of an open channel, including its eventual closure.

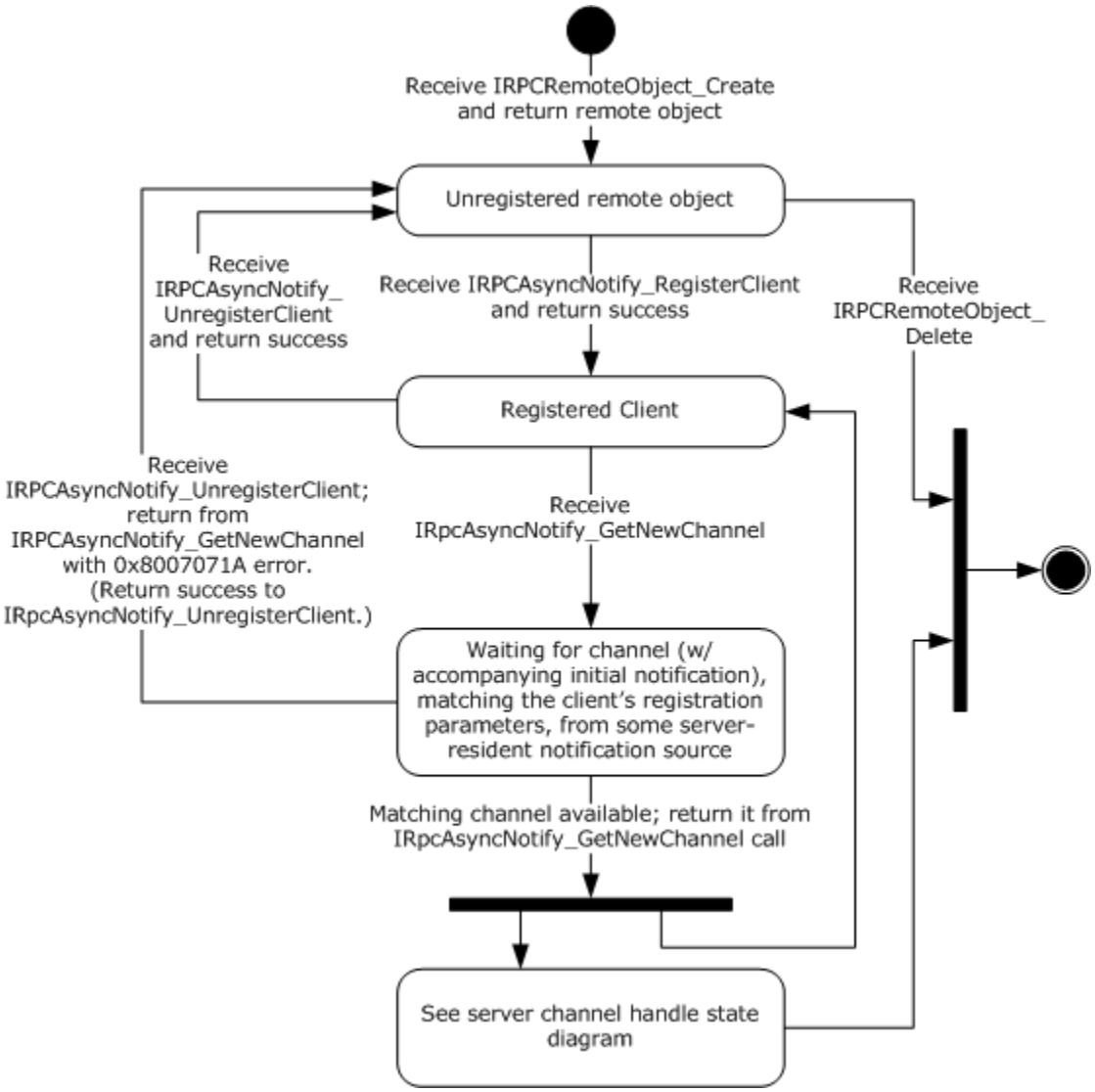


**Figure 5: Bidirectional message passing mode**

The following diagram illustrates the processing of a single open channel.

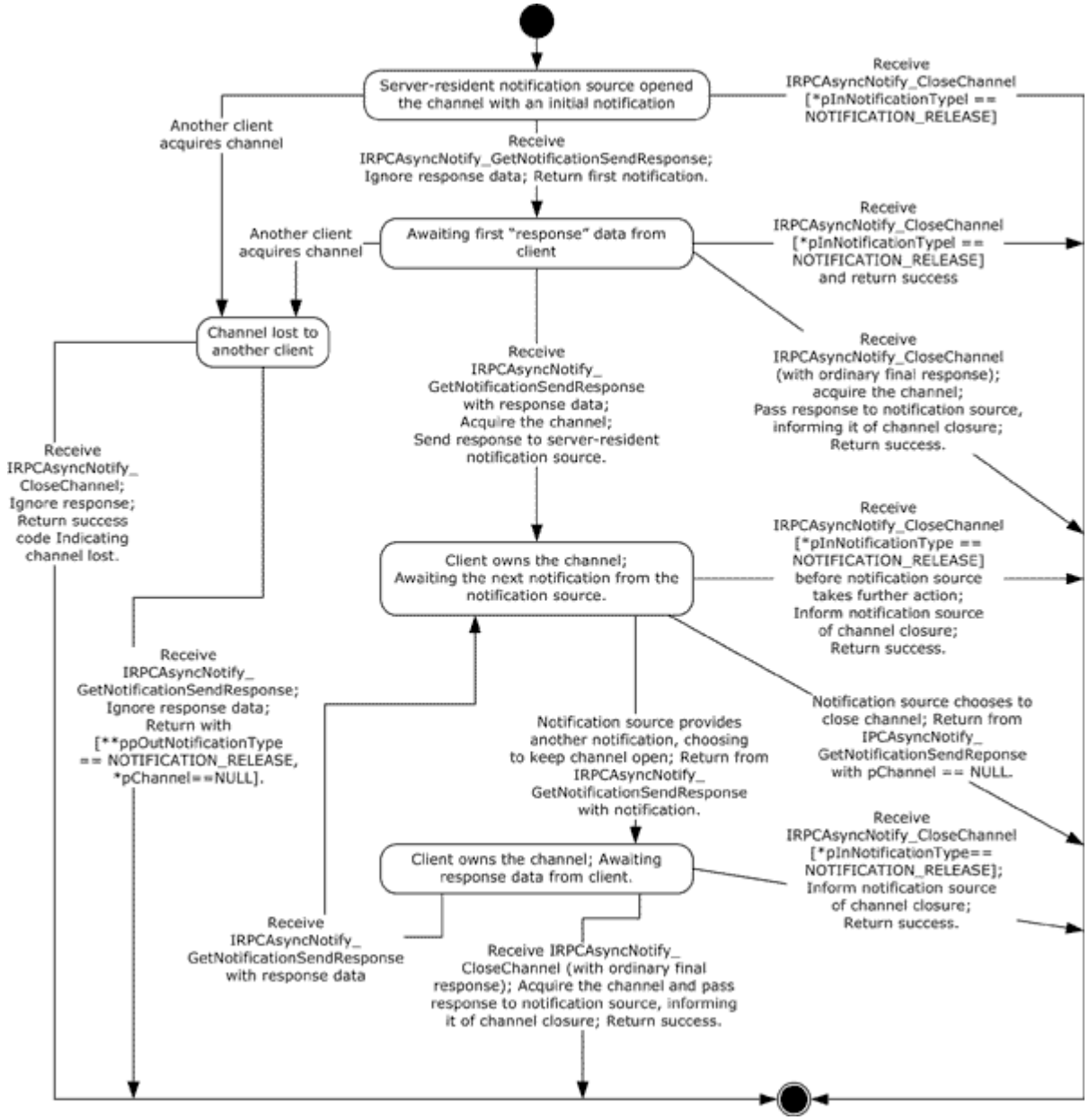


**Figure 6: Processing a single open channel**

### 3.1.1 Abstract Data Model

No abstract data model is required.

### 3.1.2 Timers

No timer events are required on the client beyond the timers required in the underlying RPC, as specified in [MS-RPCE] section 3.

### 3.1.3 Initialization

The server MUST listen on dynamically assigned endpoints, as specified in [C706] section 6.2.2.

### 3.1.4 Message Processing Events and Sequencing Rules

As specified in [MS-RPCE] section 3, this protocol MUST direct the RPC runtime to do the following:

- Perform a strict NDR data consistency check at target level 6.0.
- Reject a null unique or full pointer with a nonzero conforming value.

Methods in RPC Opnum Order

| Method | Description |
|---|---|
| **IRPCAsyncNotify_RegisterClient** | This method is called by clients to register to receive notifications, and to associate the parameters describing the set of notifications they are registering to receive, with a remote object.<br>Opnum: 0 |
| **IRPCAsyncNotify_UnregisterClient** | This method is called by registered clients to unregister remote objects.<br>Opnum: 1 |
| Opnum2NotUsedOnWire | Reserved for local use.<br>Opnum: 2 |
| **IRPCAsyncNotify_GetNewChannel** | This method returns an array of pointers to print notification channels.<br>Opnum: 3 |
| **IRPCAsyncNotify_GetNotificationSendResponse** | This method sends a client response to the server and returns the next notification sent by way of the same channel when it becomes available.<br>Opnum: 4 |
| **IRPCAsyncNotify_GetNotification** | This method returns notification data from the server.<br>Opnum: 5 |
| **IRPCAsyncNotify_CloseChannel** | This method sends a final response on the notification channel and closes it.<br>Opnum: 6 |

In the table above, the term "Reserved for local use" means that the client MUST NOT send the **opnum**, and the server behavior is undefined<29> since it does not affect interoperability.

There are additional syntax rules that apply to the common string and flag fields that are passed as parameters to methods in this protocol. These rules are specified in [MS-RPRN] section 2.2.4.

### 3.1.4.1 IRPCAsyncNotify_RegisterClient (Opnum 0)

The **IRPCAsyncNotify_RegisterClient** method is called by clients to register to receive notifications, and to associate the parameters describing the set of notifications they are registering to receive, with a remote object.

```
HRESULT IRPCAsyncNotify_RegisterClient(
  [in] PRPCREMOTEOBJECT pRegistrationObj,
  [in, string, unique] const wchar_t* pName,
  [in] PrintAsyncNotificationType* pInNotificationType,
  [in] PrintAsyncNotifyUserFilter NotifyFilter,
  [in] PrintAsyncNotifyConversationStyle conversationStyle,
  [out, string] wchar_t** ppRmtServerReferral
);
```

**pRegistrationObj:** MUST be the remote object context handle, which MUST have been returned by the server in the ppRemoteObject output parameter of a prior call to **IRPCRemoteObject_Create**. This value MUST NOT be null.

**pName:** MUST be a pointer to a null-terminated string that MUST specify the name of the printer from which the client is registering to receive notifications. If null, the registration MUST be made for the remote print server itself.

**pInNotificationType:** MUST be a pointer to a **PrintAsyncNotificationType** structure that MUST specify the notification type identifier for the notifications that the client is registering to receive.

**NotifyFilter:** MUST be a value of type **PrintAsyncNotifyUserFilter** that MUST specify whether the client is registering to receive notifications that are issued to all registered clients, irrespective of their authenticated user identity or to receive notifications that are issued only to the specific authenticated user identity of the registering RPC client.

**conversationStyle:** MUST be a value of type **PrintAsyncNotifyConversationStyle** that MUST specify whether the client is registering for bidirectional communication mode or unidirectional communication mode.

**ppRmtServerReferral:** Reserved for future use. Clients and servers SHOULD NOT<30> use this parameter.

**Return Values:** This method MUST return 0x00000000 to indicate successful completion, or an HRESULT failure value, as specified in [MS-ERREF]. <31>

The client MUST treat all error values the same; the error MUST be considered fatal and reported to the higher-level caller. <32>

**Exceptions Thrown:** No exceptions are thrown besides those thrown by the underlying RPC protocol (as specified in [MS-RPCE]).

The **IRPCAsyncNotify_RegisterClient** method MUST be called by clients to register for receiving notifications. Servers MUST associate the given remote object with the registration parameters specified.

A client MUST NOT call **IRPCAsyncNotify_RegisterClient** if a prior call to **IRPCAsyncNotify_RegisterClient** succeeded using the same PRPCREMOTEOBJECT value, unless a later call to **IRPCAsyncNotify_UnregisterClient** also succeeded.<33>

If registering for unidirectional communication mode, a client SHOULD call **IRPCAsyncNotify_GetNotification** after a successful call to **IRPCAsyncNotify_RegisterClient** using the same **PRPCREMOTEOBJECT** value.

If registering for bidirectional communication mode, a client SHOULD call **IRPCAsyncNotify_GetNewChannel** after a successful call to **IRPCAsyncNotify_RegisterClient** using the same **PRPCREMOTEOBJECT** value.

Servers MUST support the concurrent registration of multiple remote objects with different registration parameters, including notification type, filter, and communication mode.

### 3.1.4.2 IRPCAsyncNotify_UnregisterClient (Opnum 1)

The **IRPCAsyncNotify_UnregisterClient** method is called by registered clients to unregister remote objects. For this call to succeed, the remote object MUST have already successfully called **IRPCAsyncNotify_RegisterClient**.

```
HRESULT IRPCAsyncNotify_UnregisterClient(
  [in] PRPCREMOTEOBJECT pRegistrationObj
);
```

**pRegistrationObj:** MUST be the remote object context handle, which MUST have been successfully registered by a prior call to **IRPCAsyncNotify_RegisterClient**. This value MUST NOT be null.

**Return Values:** This method MUST return an HRESULT success value, as specified in [MS-ERREF], to indicate successful completion, or an HRESULT failure value otherwise. The client MUST treat all error values the same; the error MUST be considered fatal and reported to the higher-level caller.<34>

**Exceptions Thrown:** No exceptions are thrown besides those thrown by the underlying Remote Procedure Call (RPC) protocol (as specified in [MS-RPCE] section 3).

If a client call to **IRPCAsyncNotify_GetNewChannel** or **IRPCAsyncNotify_GetNotification** is blocked on the server waiting for a notification channel or notification to become available, the server MUST process a client call to **IRPCAsyncNotify_UnregisterClient** without waiting for the notification channel or notification.

A server MUST NOT:

- Indicate success to a client call of **IRPCAsyncNotify_UnregisterClient**, unless a prior call to **IRPCAsyncNotify_RegisterClient** succeeded using the same **PRPCREMOTEOBJECT** value; or
- Indicate success to a client call of **IRPCAsyncNotify_UnregisterClient**, following a prior successful call to **IRPCAsyncNotify_UnregisterClient** by using the same **PRPCREMOTEOBJECT** value.

### 3.1.4.3 IRPCAsyncNotify_GetNewChannel (Opnum 3)

The **IRPCAsyncNotify_GetNewChannel** method returns an array of pointers to print notification channels. This method MUST only be used with bidirectional communication mode.

```
HRESULT IRPCAsyncNotify_GetNewChannel(
  [in] PRPCREMOTEOBJECT pRemoteObj,
  [out] unsigned long* pNoOfChannels,
```

```
  [out, size_is(, *pNoOfChannels)]
    PNOTIFYOBJECT** ppChannelCtxt
);
```

**pRemoteObj:** MUST be the remote object context handle. This handle is obtained from **IRPCRemoteObject_Create**. This remote object MUST have been registered for bidirectional communication mode by a prior successful call to **IRPCAsyncNotify_RegisterClient**.

**pNoOfChannels:** MUST specify the number of notification channels returned. The array of notification channels is specified by the **ppChannelCtxt** parameter.<35>

**ppChannelCtxt:** MUST specify a pointer to the array of returned notification channels.

**Return Values:** This method MUST return 0 to indicate successful completion, or an HRESULT failure value otherwise, as specified in [MS-ERREF]. Protocol-specific error values are defined in the following table. The client MUST treat all other error values the same, and MAY<36> retry after receiving such an error value.

| Return value | Description |
|---|---|
| 0x8004000C | Indicates that a prior call to this method, which specified the same remote object, MUST complete before the server can process additional calls to this method for that object.<br>The client SHOULD NOT retry the call before the previous call to this method for the specified remote object has completed. |
| 0x8004000E | Indicates that the specified remote object was already deregistered by a prior call to **IRPCAsyncNotify_UnregisterClient**.<br>The client SHOULD NOT retry the call. <37> |
| 0x80070057 | Indicates that the specified remote object is in an invalid state due to a prior failed call to **IRPCAsyncNotify_RegisterClient**.<br>The client SHOULD NOT retry the call. |
| 0x8007071A | Indicates termination of incoming notifications. Upon completion of this call with this return value, the server MUST fail incoming calls to **IRPCAsyncNotify_GetNewChannel** by using the same value of pRemoteObj parameter.<br>The client SHOULD NOT retry the call. |

**Exceptions Thrown:** An exception code of 0x08004000C, 0x8004000E or 0x8007071A MAY<38> be thrown by the server under the circumstances described in the preceding table for the corresponding return values. The client MUST treat these exception codes exactly as it would treat the same return values. No additional exceptions are thrown aside from those thrown by the underlying RPC protocol (as specified in [MS-RPCE] section 3).

If successful, the **IRPCAsyncNotify_GetNewChannel** method returns an array of pointers to print notification channels.

A server MUST NOT:

- Indicate success to a client call of **IRPCAsyncNotify_GetNewChannel** unless a prior call to **IRPCAsyncNotify_RegisterClient** succeeded using the same **PRPCREMOTEOBJECT** value.

- Indicate success to a client call of **IRPCAsyncNotify_GetNewChannel** following a prior successful call to **IRPCAsyncNotify_UnregisterClient** using the same **PRPCREMOTEOBJECT** value.
- Complete calls to **IRPCAsyncNotify_GetNewChannel** unless a server event has occurred, such as:
  - A notification source opened a new channel.
  - An abnormal event happened, such as an initiated server shutdown sequence.

A client SHOULD:

- Call **IRPCAsyncNotify_GetNewChannel** in response to a prior successful return from **IRPCAsyncNotify_RegisterClient** or **IRPCAsyncNotify_GetNewChannel**.<39>
- Call **IRPCAsyncNotify_GetNotificationSendResponse** in response to a prior successful return from **IRPCAsyncNotify_GetNewChannel**.<40>

A client MUST NOT:

- Call **IRPCAsyncNotify_GetNewChannel**, unless a prior call to **IRPCAsyncNotify_RegisterClient** succeeded by using the same **PRPCREMOTEOBJECT** value.
- Call **IRPCAsyncNotify_GetNewChannel**, following a prior call to **IRPCAsyncNotify_UnregisterClient** by using the same **PRPCREMOTEOBJECT** value.

#### 3.1.4.4 IRPCAsyncNotify_GetNotificationSendResponse (Opnum 4)

The **IRPCAsyncNotify_GetNotificationSendResponse** method sends a client response to the server and returns the next notification sent by way of the same channel when it becomes available. This method MUST be used only with bidirectional communication mode.

```
HRESULT IRPCAsyncNotify_GetNotificationSendResponse(
  [in, out] PNOTIFYOBJECT* pChannel,
  [in, unique] PrintAsyncNotificationType* pInNotificationType,
  [in] unsigned long InSize,
  [in, size_is(InSize), unique] byte* pInNotificationData,
  [out] PrintAsyncNotificationType** ppOutNotificationType,
  [out] unsigned long* pOutSize,
  [out, size_is(,*pOutSize)] byte** ppOutNotificationData
);
```

**pChannel:** A client MUST provide a pointer to a notification channel that MUST NOT be closed or zero, and which MUST have been returned by the server in the ppChannelCtxt output parameter of a prior call to **IRPCAsyncNotify_GetNewChannel**. If the server closes the notification channel, it MUST set the **pChannel** value to null upon return from this method. If the notification channel was acquired by a different client, the server MUST set the **pChannel** value to null upon return from this method.

**pInNotificationType:** Must be a pointer to a **PrintAsyncNotificationType** structure that MUST specify the notification type identifier of the notification type that the registered client is interested in. The value MUST be the same as the notification type identifier of the notification type for which the client has registered.

**InSize:** MUST be the length of client-to-server response data, in number of bytes. The server MAY<41> impose an upper limit on this value that is smaller than the maximum unsigned 32-bit integer.

**pInNotificationData:** A pointer to input data holding the client's response to the previous notification received on the same bidirectional notification channel.

On the first call to **IRPCAsyncNotify_GetNotificationSendResponse** for a given channel, the client SHOULD provide zero bytes of response data, and the server MUST ignore any response data sent.

On subsequent calls to **IRPCAsyncNotify_GetNotificationSendResponse**, the response format MUST conform to the requirements of the notification channel's notification type, and those notification type requirements determine whether or a not a zero-byte response is acceptable.

If the value of **InSize** is not 0x00000000, then **pInNotificationData** MUST NOT be null.

**ppOutNotificationType:** MUST return a pointer to the notification type identifier of the of server-to-client notification. If the notification channel was acquired by a different client, the value MUST be **NOTIFICATION_RELEASE** (for more information, see section 2.2.1). If the server needs to close the notification channel without sending a final response, the value SHOULD be **NOTIFICATION_RELEASE**. In all other cases, the value MUST be the same as the notification type identifier of the notification type for which the client has registered.

**pOutSize:** MUST be the length of server-to-client notification data, in number of bytes. The client MAY<42> impose an upper limit on this value that is smaller than the maximum unsigned 32-bit integer. If the notification channel was acquired by a different client, the server SHOULD set the value of **pOutSize** to 0x00000000. If the value of **ppOutNotificationType** points to **NOTIFICATION_RELEASE**, the server SHOULD set the value of **pOutSize** to 0x00000000.

**ppOutNotificationData:** MUST return a pointer to server-to-client notification data in a format that MUST conform to the notification channel's notification type. If the notification channel was acquired by a different client, the server SHOULD set the value of **ppOutNotificationData** to null. If the value of **ppOutNotificationType** points to **NOTIFICATION_RELEASE**, the client MUST ignore the content of **ppOutNotificationData**.

**Return Values:** This method MUST return 0x00000000 upon successful completion or an HRESULT failure value, as specified in [MS-ERREF]. The client MUST treat all error values the same; the error MUST be considered fatal and reported to the higher-level caller.<43>

**Exceptions Thrown:** No exceptions are thrown aside from those thrown by the underlying Remote Procedure Call Protocol, as specified in [MS-RPCE] section 3.

The first call to this method on the newly opened notification channel serves as a mediator among all the clients that registered themselves for the given notification type. This MUST be done by blocking all calls from clients until a matching server-side event occurs, including the following:

- The channel issues a notification.
- An abnormal condition occurs, such as an initiated server shutdown sequence.
- The server receives a client request to close the channel.

The server MUST:

- Choose the first client that sent a response, whether by calling this method or by calling IRPCAsyncNotify_CloseChannel with a notification type identifier other than NOTIFICATION_RELEASE, and assign the opened notification channel to that client.
- For all other clients, set the value of the **ppOutNotificationType** output parameter to NOTIFICATION_RELEASE and the value of the **pChannel** parameter to null.
- Return an HRESULT success value, as specified in [MS-ERREF], to all the other clients that have outstanding blocked calls to this method.

All subsequent calls to this method MUST take the response provided by the client that was assigned to the notification channel, and pass it to the server-resident notification source that opened the notification channel. The call MUST return when a subsequent notification is available, the channel is closed, or an abnormal event happens, such as the print spooler server terminating its execution.

The server MUST NOT indicate success to a client call to this method if a prior call to IRPCAsyncNotify_CloseChannel succeeded, specifying the same notification channel.

A client MUST NOT call **IRPCAsyncNotify_GetNotificationSendResponse** following a prior successful return from **IRPCAsyncNotify_GetNotificationSendResponse** with a null output value of the **pChannel** parameter, or following a prior successful return from **IRPCAsyncNotify_CloseChannel**.

A client SHOULD call **IRPCAsyncNotify_GetNotificationSendResponse** or **IRPCAsyncNotify_CloseChannel**, following a prior successful return from **IRPCAsyncNotify_GetNotificationSendResponse** with a non-null output value of the **pChannel** parameter.

### 3.1.4.5 IRPCAsyncNotify_GetNotification (Opnum 5)

The **IRPCAsyncNotify_GetNotification** method returns notification data from the server. This method MUST NOT be used with bidirectional communication mode.

```
HRESULT IRPCAsyncNotify_GetNotification(
  [in] PRPCREMOTEOBJECT pRemoteObj,
  [out] PrintAsyncNotificationType** ppOutNotificationType,
  [out] unsigned long* pOutSize,
  [out, size_is(,*pOutSize)] byte** ppOutNotificationData
);
```

**pRemoteObj:** MUST be the remote object context handle. This remote object MUST have been registered for unidirectional communication mode by a prior successful call to **IRPCAsyncNotify_RegisterClient**.

**ppOutNotificationType:** MUST return a pointer to the notification type identifier of the server-to-client notification. If the registered remote object has been deleted, the value MUST be NOTIFICATION_RELEASE (for more information, see section 2.2.1). In all other cases the value MUST be the same as the notification type identifier of the notification type for which the client has registered.

**pOutSize:** MUST be the length of server-to-client notification data, in number of bytes. The client MAY<44> impose an upper limit on this value that is smaller than the maximum unsigned 32-bit integer.

**ppOutNotificationData:** MUST be a pointer to server-to-client notification data in a format that MUST conform to the channel's notification type.

**Return Values:** This method MUST return 0 upon successful completion or an HRESULT failure value, as specified in [MS-ERREF]. Protocol-specific error codes are defined in the following table. The client MUST treat all other error values the same, and MAY<45> retry after receiving such an error value.

| Return value | Description |
|---|---|
| 0x8004000C | Indicates that a prior call to this method, which specified the same remote object, MUST complete before the server can process additional calls to this method for that object.<br>The client SHOULD NOT retry the call unless the previous call to this method for the specified remote object has completed. |
| 0x8004000E | Indicates that the specified remote object was already deregistered by a prior call to **IRPCAsyncNotify_UnregisterClient**.<br>The client SHOULD NOT retry the call.<46> |
| 0x80070057 | Indicates that the specified remote object is in an invalid state due to a prior failed call to **IRPCAsyncNotify_RegisterClient**.<br>The client SHOULD NOT retry the call.<47> |
| 0x8007071A | Indicates the termination of incoming notifications. Upon completion of this call with this return value, the server MUST fail incoming calls to **IRPCAsyncNotify_GetNotification** using the same value of pRemoteObj parameter.<br>The client SHOULD NOT retry the call. |

**Exceptions Thrown:** An exception code of 0x08004000C, 0x8004000E or 0x8007071A MAY<48> be thrown by the server, under the circumstances described in the preceding table for the corresponding return values. The client MUST treat these exception codes exactly as it would treat the same return values. No additional exceptions are thrown beyond those thrown by the underlying Remote Procedure Call (RPC) protocol, as specified in [MS-RPCE] section 3.

The **IRPCAsyncNotify_GetNotification** method MUST return data from the server that matches the registration for the given remote object.

A server MUST NOT:

- Indicate success to a client call of **IRPCAsyncNotify_GetNotification** unless a prior call to **IRPCAsyncNotify_RegisterClient** succeeded using the same **PRPCREMOTEOBJECT** value.<49>
- Indicate success to a client call of **IRPCAsyncNotify_GetNotification** following a prior successful call to **IRPCAsyncNotify_UnregisterClient** using the same **PRPCREMOTEOBJECT** value.
- Complete this call until there is data that can be returned to the client or an abnormal condition occurs—for example, an initiated server shutdown sequence or remote object unregistration—that will result in a failure error code returned prior to the server having data.

If a client wants to receive further notifications from the server, the client SHOULD <50>call **IRPCAsyncNotify_GetNotification** in response to a prior successful return from

**IRPCAsyncNotify_GetNotification**. When the client no longer wants to receive notifications from the server, it SHOULD call **IRPCAsyncNotify_UnregisterClient**, either before or after the return from **IRPCAsyncNotify_GetNotification**.

A client MUST NOT:

- Call **IRPCAsyncNotify_GetNotification** unless a prior call to **IRPCAsyncNotify_RegisterClient** succeeded, using the same PRPCREMOTEOBJECT value; or
- Call **IRPCAsyncNotify_GetNotification** following a prior call to **IRPCAsyncNotify_UnregisterClient**, by using the same **PRPCREMOTEOBJECT** value.

### 3.1.4.6 IRPCAsyncNotify_CloseChannel (Opnum 6)

The **IRPCAsyncNotify_CloseChannel** method sends a final response on the notification channel and closes it. This method MUST NOT be used with unidirectional communication mode.

```
HRESULT IRPCAsyncNotify_CloseChannel(
  [in, out] PNOTIFYOBJECT* pChannel,
  [in] PrintAsyncNotificationType* pInNotificationType,
  [in] unsigned long InSize,
  [in, size_is(InSize), unique] byte* pReason
);
```

**pChannel:** MUST be a pointer to a notification channel, which MUST NOT be closed or zero, and which MUST have been returned by the server in the **ppChannelCtxt** output parameter of a prior call to **IRPCAsyncNotify_GetNewChannel**. Upon receipt, the server MUST set the **pChannel** value to null.

**pInNotificationType:** MUST be a pointer to a **PrintAsyncNotificationType** value. If the client needs to close the notification channels without sending a final response, this value SHOULD point to **NOTIFICATION_RELEASE**. In all other cases, this value MUST point to the notification type identifier of the notification type for which the client has registered.

**InSize:** MUST be the length of client-to-server notification data, in number of bytes. The server MAY<51> impose an upper limit on this value that is smaller than the maximum unsigned 32-bit integer.

If **pInNotificationType** is **NOTIFICATION_RELEASE**, InSize SHOULD be 0x00000000.

**pReason:** MUST be a pointer to a sequence of bytes conveying final client-to-server response data. The number of bytes MUST be provided in the InSize parameter. If InSize is not 0x00000000, then pReason MUST NOT be null.

If **pInNotificationType** is **NOTIFICATION_RELEASE**, the client SHOULD provide zero bytes of response data, and the server MUST ignore any response data pointed to by **pReason**. If **pInNotificationType** is not **NOTIFICATION_RELEASE**, the response format MUST conform to the requirements of the notification channel's notification type, and those notification type requirements determine whether or a not a zero-byte response is acceptable.

**Return Values:** This method MUST return an HRESULT value to indicate either success or failure, as specified in [MS-ERREF]. The client MUST consider all error return values fatal and report them to the higher-level caller.<52>

**Exceptions Thrown:** No exceptions are thrown besides those thrown by the underlying Remote Procedure Call Protocol, specified in [MS-RPCE].

If a client call to **IRPCAsyncNotify_GetNotificationSendResponse** is blocked on the server, waiting for a notification to become available on a notification channel, the server MUST process a client call to this method on the same notification channel without waiting for a notification.

A client MUST NOT call **IRPCAsyncNotify_CloseChannel** following a prior successful return from **IRPCAsyncNotify_GetNotificationSendResponse** with a null value of **pChannel** parameter, or following a prior successful return from **IRPCAsyncNotify_CloseChannel**.

### 3.1.5 Timer Events

No timer events are required on the server beyond the timers required in the underlying Remote Procedure Call (RPC) protocol, as specified in [MS-RPCE] section 3.

### 3.1.6 Other Local Events

This protocol does not define the set of printing events that cause notification sources to trigger notifications.

## 3.2 Asynchronous Notification Client Details

Unidirectional message passing mode is illustrated by the following client state diagram. This diagram represents a client registering and receiving notifications of a predetermined notification type and user filter, as specified in the **IRPCAsyncNotify_RegisterClient** parameters (for more information, see section 3.1.4.1).

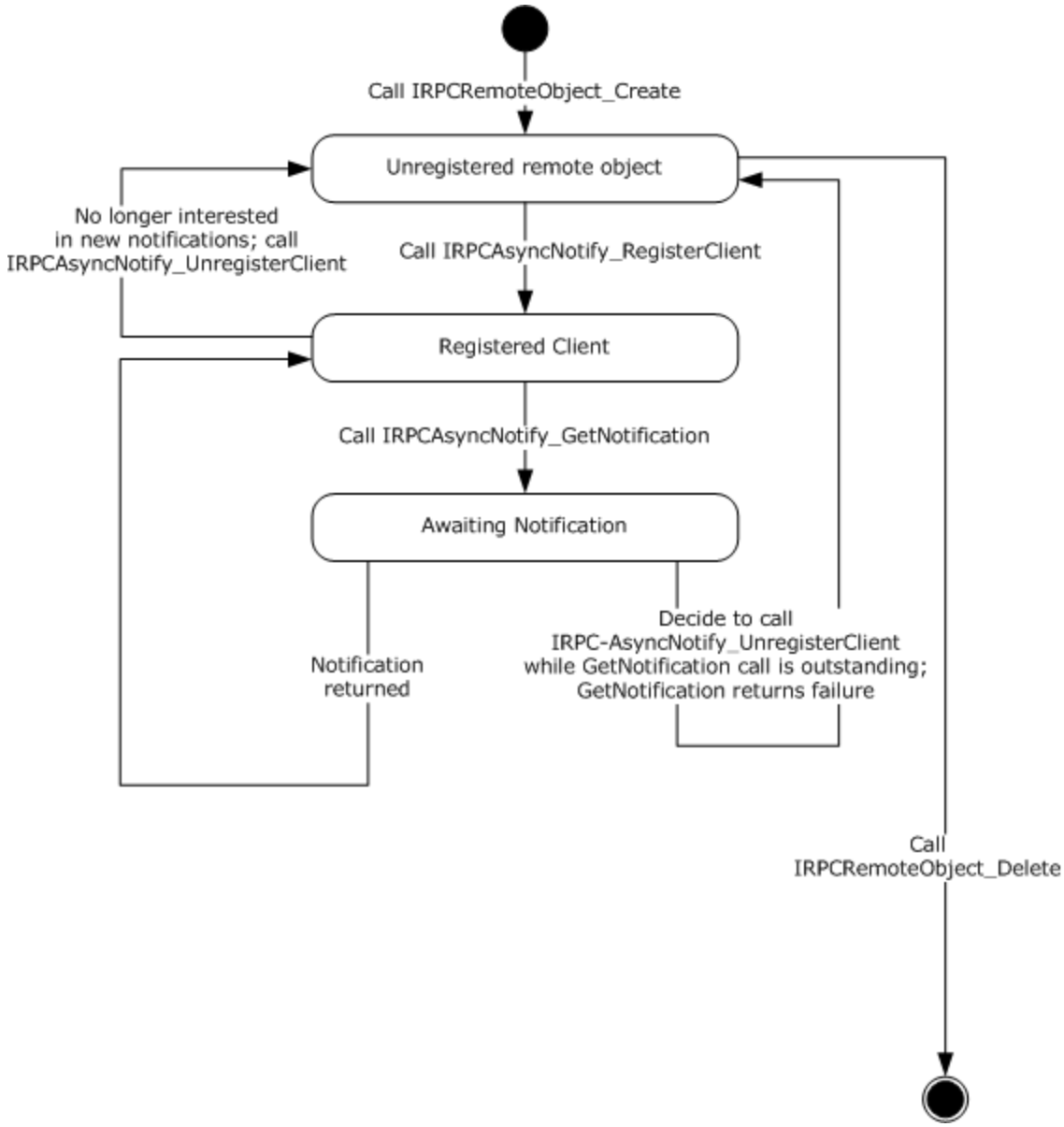


**Figure 7: Client registering and receiving notifications of a predetermined notification type**

Bidirectional message passing mode is illustrated by the following two client state diagrams. The first diagram illustrates remote object creation and deletion, client registration, and the opening of notification channels. The second diagram provides the details of the processing of an open channel, including its eventual closure.

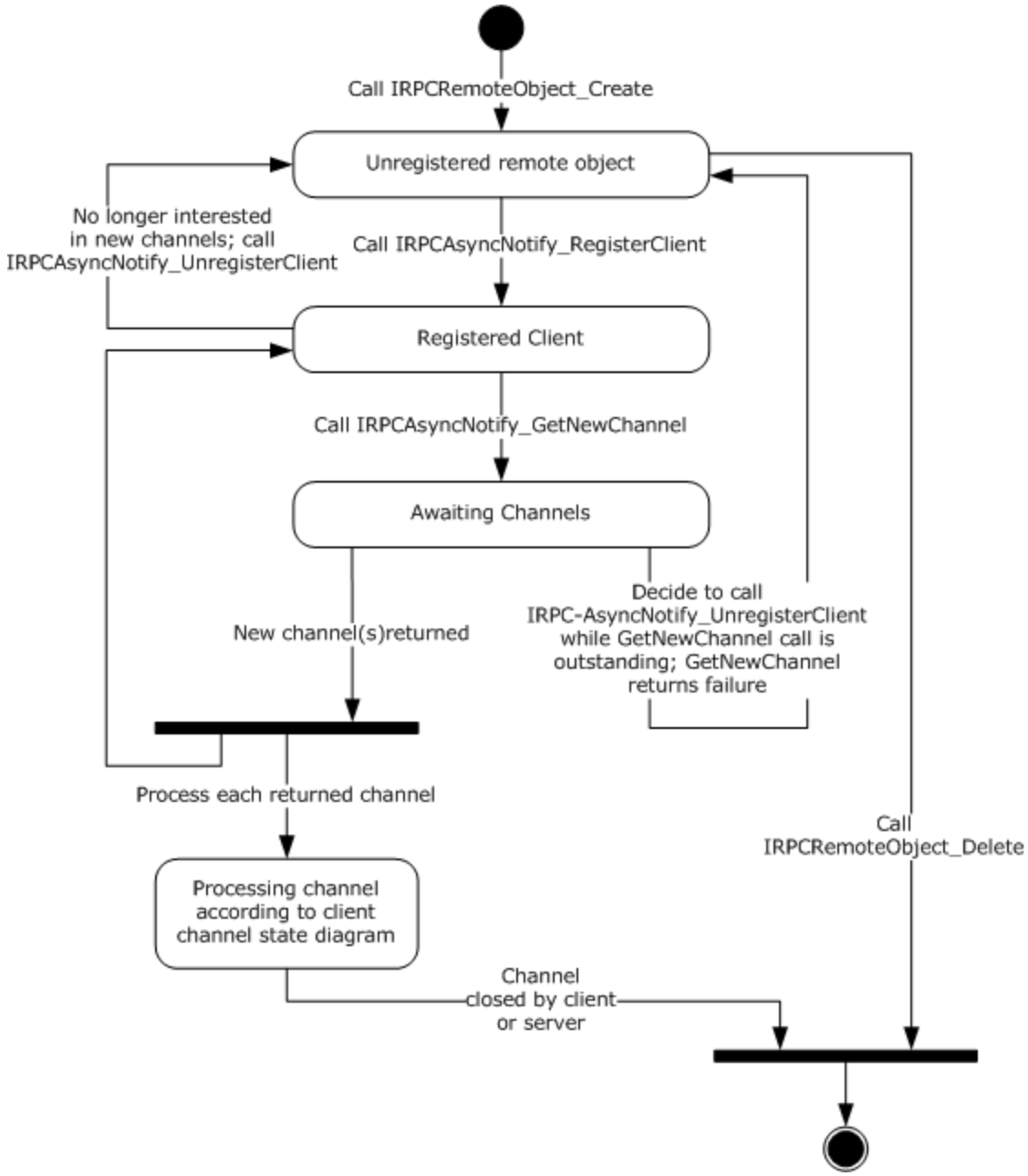


**Figure 8: Remote object creation and deletion, client registration, and opening of notification channels**

The following diagram illustrates the processing of a single open channel.

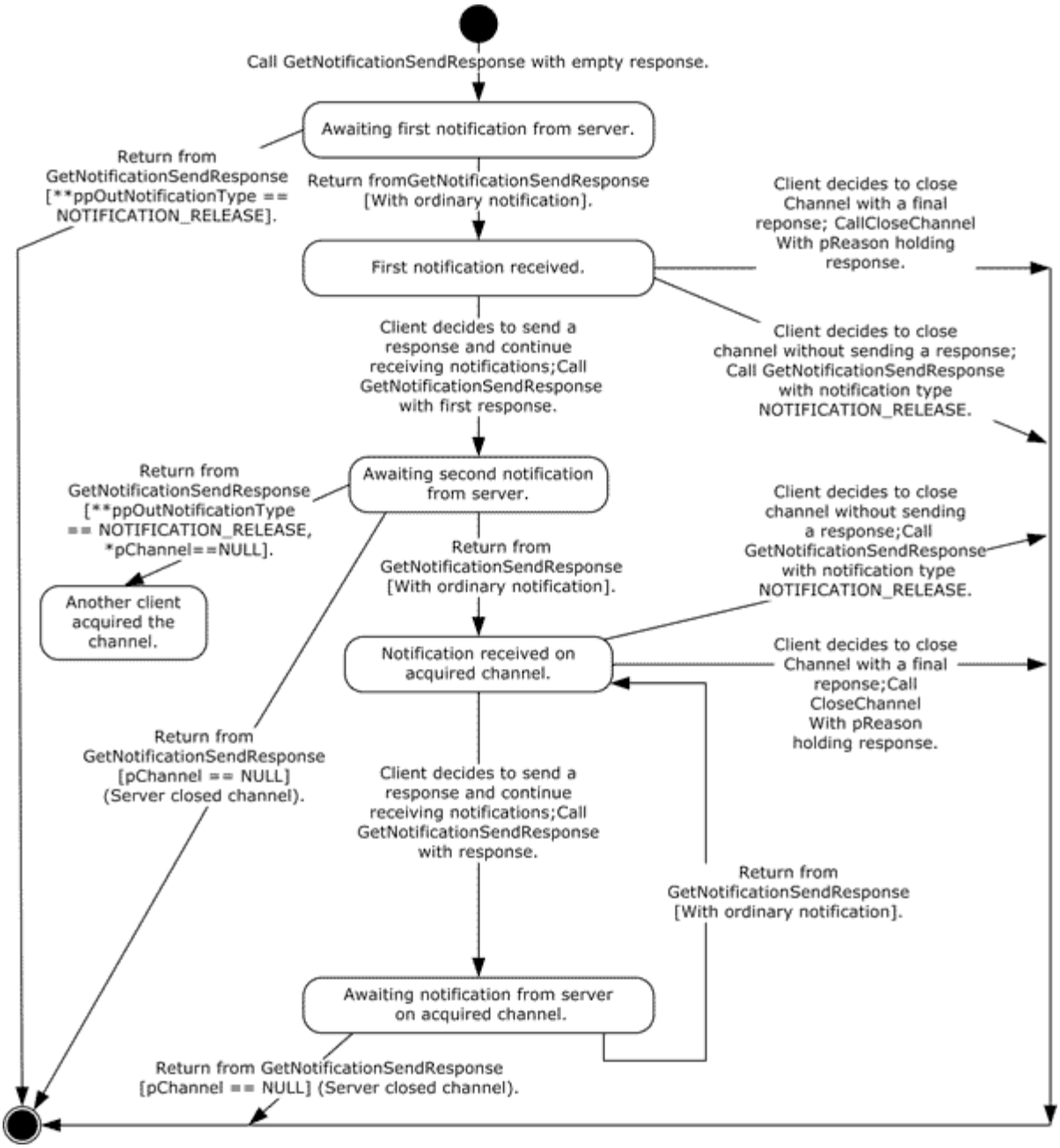


**Figure 9: Processing a single open channel**

### 3.2.1 Abstract Data Model

No abstract data model is required.

### 3.2.2 Timers

No timers are required beyond those used internally by RPC to implement resiliency to network outages, as specified in [MS-RPCE] section 3. A timer MAY<53> be used for reconnection attempts.

### 3.2.3 Initialization

The client MUST create an RPC binding handle to the server RPC dynamic endpoint when an RPC method is called, as specified in [C706] section 2.3.

### 3.2.4 Message Processing Events and Sequencing Rules

This protocol MUST direct the RPC runtime to:

- Perform a strict NDR data consistency check at target level 6.0; and
- Reject a null unique or full pointer with a non-zero conforming value;

as specified in [MS-RPCE] section 3.

Clients MUST manage registrations throughout their lifetimes. Specifically, clients MUST call **IRPCAsyncNotify_UnregisterClient** for each successful call to **IRPCAsyncNotify_RegisterClient**.

When either **IRPCAsyncNotify_GetNewChannel** or **IRPCAsyncNotify_GetNotification** returns with a success code, the client SHOULD issue the next call of the same kind as soon as possible, to minimize the amount of buffering and risk of event loss on the server.

The syntax and behavior for the methods of the IRPCAsyncNotify interface are fully specified in section 3.1.4.

### 3.2.5 Timer Events

No timer events are required on the client except the timers that are required in the underlying Remote Procedure Call (RPC) protocol, as specified in [MS-RPCE] section 3.

### 3.2.6 Other Local Events

A client's registration is typically the result of printing activity.

## 3.3 Remote Object Server Details

The remote object server interface provides methods that allow a client to define a generic remote object on a server. The version for this interface is 1.0. To receive incoming remote calls for this interface, the client MUST establish an RPC dynamic endpoint by using the Universal Unique Identifier (UUID) "ae33069b-a2a8-46ee-a235-ddfd339be281".

### 3.3.1 Abstract Data Model

No abstract data model is required.

### 3.3.2 Timers

No protocol timer events are required on the client beyond the timers required in the underlying Remote Procedure Call (RPC) protocol, as specified in [MS-RPCE] section 3.

### 3.3.3 Initialization

The server MUST listen on dynamically-assigned endpoints, as specified in [C706], section 6.2.2.

### 3.3.4 Message Processing Events and Sequencing Rules

This protocol MUST direct the RPC runtime to:

- Perform a strict NDR data consistency check at target level 6.0; and
- Reject a null unique or full pointer with non-zero conforming value;

as specified in [MS-RPCE] section 3.

The sections that follow specify the syntax and behavior for each method defined in this interface:

Methods in RPC Opnum Order

| Method | Description |
|---|---|
| **IRPCRemoteObject_Create** | The method creates a remote object on a server and returns it to the client.<br>Opnum: 0 |
| **IRPCRemoteObject_Delete** | The method destroys the specified remote object.<br>Opnum: 1 |

There are additional syntax rules that apply to the common string and flag fields passed as parameters to methods in this protocol. These rules are specified in [MS-RPRN] section 2.2.4.

#### 3.3.4.1 IRPCRemoteObject_Create (Opnum 0)

The **IRPCRemoteObject_Create** method creates a remote object on a server and returns it to the client.

```
HRESULT IRPCRemoteObject_Create(
  [in] handle_t hRemoteBinding,
  [out] PRPCREMOTEOBJECT* ppRemoteObj
);
```

**hRemoteBinding:** MUST be a client-generated RPC binding handle by using a **Universal Naming Convention (UNC)** name, which MUST uniquely identify a print server on the network. RPC binding handles are as specified in [C706] section 2.3.

**ppRemoteObj:** MUST be a remote object context handle returned by the server. It MUST be a non-null value.

**Return Values:** This method MUST return 0 to indicate successful completion, or an HRESULT failure value otherwise, as specified in [MS-ERREF]. The client MUST treat all unsuccessful return values as fatal errors.

**Exceptions Thrown:** No exceptions are thrown aside from those thrown by the underlying Remote Procedure Call (RPC) protocol, as specified in [MS-RPCE] section 3.

#### 3.3.4.2 IRPCRemoteObject_Delete (Opnum 1)

The **IRPCRemoteObject_Delete** method destroys the specified remote object.

```
void IRPCRemoteObject_Delete(
  [in, out] PRPCREMOTEOBJECT* ppRemoteObj
);
```

**ppRemoteObj:** MUST be the remote object to delete. The handle MUST have been returned by the server in the ppRemoteObj output parameter of a prior call to **IRPCRemoteObject_Create**, and MUST NOT have been previously deleted. If this handle was previously registered by a successful call to **IRPCAsyncNotify_RegisterClient**, then it MUST have been subsequently unregistered by a call to **IRPCAsyncNotify_UnregisterClient**. It MUST NOT be null.

Upon receipt, the server MUST set the ppRemoteObj value to null.

**Return Values:** This method has no return values.

**Exceptions Thrown:** No exceptions are thrown beyond those thrown by the underlying Remote Procedure Call Protocol, as specified in [MS-RPCE] section 3.

### 3.3.5 Timer Events

No protocol timer events are required on the server beyond the timers required in the underlying Remote Procedure Call (RPC) protocol, as specified in [MS-RPCE] section 3.

### 3.3.6 Other Local Events

No high-level triggered events are processed.

## 3.4 Remote Object Client Details

### 3.4.1 Abstract Data Model

No abstract data model is required.

### 3.4.2 Timers

No timers are required beyond those used internally by RPC to implement resiliency to network outages, as specified in [MS-RPCE] section 3. A timer MAY<54> be used for reconnection attempts.

### 3.4.3 Initialization

The client creates an RPC binding handle, as specified in [C706] section 2.3, to the server RPC dynamic endpoint when an RPC method is called.

### 3.4.4 Message Processing Events and Sequencing Rules

This protocol MUST direct the RPC runtime to:

- Perform a strict NDR data consistency check at target level 6.0; and
- Reject a NULL unique or full pointer with a non-zero conforming value;

as specified in [MS-RPCE] section 3.

Remote object clients MUST manage the lifetime of the remote objects. Specifically, clients MUST call **IRPCRemoteObject_Delete** for each successful call to **IRPCRemoteObject_Create**. These methods are specified in section 3.3.4.

### 3.4.5 Timer Events

No protocol timer events are required on the client beyond the timers required in the underlying Remote Procedure Call (RPC) protocol, as specified in [MS-RPCE] section 3.

### 3.4.6 Other Local Events

A client's invocation of each method is typically the result of local application activity. No other higher-layer triggered events are processed.

## 3.5 AsyncUI Server Details

The AsyncUI notification type allows for a print-server-resident notification source to request the following:

- Display of an informative alert on a client machine.
- That the client send user input requested by such an alert back to the server.
- That the client execute code that is resident on the client machine.

The AsyncUI notification type MUST use the notification type identifier value **AsyncPrintNotificationType_AsyncUI**, as specified in section 2.2.1.

### 3.5.1 Abstract Data Model

No abstract data model is required.

### 3.5.2 Timers

No timer events are required on the client besides the timers required in the underlying Remote Procedure Call (RPC) protocol, as specified in [MS-RPCE] section 3.

### 3.5.3 Initialization

A remote object server (for more information, see section 3.3), and an asynchronous notification server (for more information, see section 3.1) MUST be fully initialized on the server.

### 3.5.4 Message Processing Events and Sequencing Rules

This protocol MUST specify a notification type identifier value **AsyncPrintNotificationType_AsyncUI** (for more information, see section 2.2.1) when registering for notifications, or requesting or returning notification or response data, using the methods of the Print System Asynchronous Notification Protocol (for more information, see section 3.1.4).

The sections that follow specify the AsyncUI-specific server syntax and behavior for each method specified in section 3.1 for the IRPCAsyncNotify interface:

Methods in RPC Opnum Order

| Method | Description |
|---|---|
| IRPCAsyncNotify_RegisterClient | The method is specified in section 3.1.4.1. Additional AsyncUI-specific server parameter details are defined here.<br>Opnum: 0 |
| IRPCAsyncNotify_UnegisterClient | The method is specified in section 3.1.4.2. There are no additional AsyncUI-specific server details for this method.<br>Opnum: 1 |
| Opnum2NotUsedOnWire | Reserved for local use<br>Opnum: 2 |
| IRPCAsyncNotify_GetNewChannel | The method is specified in section 3.1.4.3. There are no additional AsyncUI-specific server details for this method.<br>Opnum: 3 |
| IRPCAsyncNotify_GetNotificationSendResponse | The method is specified in section 3.1.4.4. Additional AsyncUI-specific server parameter details are defined here.<br>Opnum: 4 |
| IRPCAsyncNotify_GetNotification | The method is specified in section 3.1.4.5. Additional AsyncUI-specific server parameter details are defined here.<br>Opnum: 5 |
| IRPCAsyncNotify_CloseChannel | The method is specified in section 3.1.4.6. Additional AsyncUI-specific server parameter details and are defined here.<br>Opnum: 6 |

In the following table, the term "Reserved for local use" means that the client MUST NOT send the opnum, and the server behavior is undefined<55> since it does not affect interoperability.

The specific notification and response formats referenced in this section are specified in section 2.3.

There is no AsyncUI-specific syntax or behavior for IRPCRemoteObject interface methods, which are specified in section 3.3.

### 3.5.4.1 IRPCAsyncNotify_RegisterClient (Opnum 0)

The IRPCAsyncNotify_RegisterClient method is specified in section 3.1.4.1. Additional AsyncUI-specific server parameter details are defined here.

**pInNotificationType:** MUST hold the notification type identifier value **AsyncPrintNotificationType_AsyncUI**, as specified in section 2.2.1.

### 3.5.4.2 IRPCAsyncNotify_UnregisterClient (Opnum 1)

The IRPCAsyncNotify_UnegisterClient method is specified in section 3.1.4.2. There are no additional AsyncUI-specific server details for this method.

### 3.5.4.3 IRPCAsyncNotify_GetNewChannel (Opnum 3)

The IRPCAsyncNotify_GetNewChannel method is specified in section 3.1.4.3. There are no additional AsyncUI-specific server details for this method.

### 3.5.4.4 IRPCAsyncNotify_GetNotificationSendResponse (Opnum 4)

The IRPCAsyncNotify_GetNotificationSendResponse method is specified in section 3.1.4.4. Additional AsyncUI-specific server parameter details are defined here.

**pInNotificationType:** MUST hold the notification type identifier value **AsyncPrintNotificationType_AsyncUI**, as specified in section 2.2.1.

**InSize:** SHOULD be 0. As specified in section 3.1.4.4, on the first IRPCAsyncNotify_GetNotificationSendResponse call using a given notification channel, the client SHOULD provide 0 bytes of data, and the server MUST ignore any response data. The AsyncUI notification type requires that any response to the initial notification sent on a bidirectional notification channel MUST be sent using a call to **IRPCAsyncNotify_CloseChannel**, and MUST NOT be sent by using a call to **IRPCAsyncNotify_GetNotificationSendResponse**.

**pInNotificationData:** SHOULD be null, consistent with the guidance of section 3.1.4.4 and the specification of the **InSize** parameter.

**ppOutNotificationData:** MUST hold server-to-client notification data, which conforms to the format for **AsyncUIMessageBox**, **AsyncUICustomUI**, or **AsyncUICustomData**, which are specified in sections 2.3.3, 2.3.5, and 2.3.7, respectively. If the notification is of format **AsyncUICustomUI** or **AsyncUICustomData**, the value of the **bidi** attribute of the contained "customUI" or "customData" element, specified in sections 2.3.5.1 and 2.3.7.1, respectively, MUST be "true".

### 3.5.4.5 IRPCAsyncNotify_GetNotification (Opnum 5)

The IRPCAsyncNotify_GetNotification method is specified in section 3.1.4.5. Additional AsyncUI-specific server parameter details are defined here.

**ppOutNotificationType:** MUST hold either notification type identifier value **AsyncPrintNotificationType_AsyncUI** or **NOTIFICATION_RELEASE**, as specified in section 2.2.1.

**ppOutNotificationData:** MUST hold server-to-client notification data, which conforms to the data format for **AsyncUIBalloon**, **AsyncUICustomUI**, or **AsyncUICustomData**, as specified in sections 2.3.2, 2.3.5, and 2.3.7, respectively. If the notification is of format **AsyncUICustomUI** or **AsyncUICustomData**, the value of the **bidi** attribute of the contained "customUI" or "customData" element, as specified in sections 2.3.5.1 and 2.3.7.1, respectively, MUST be "false".

### 3.5.4.6 IRPCAsyncNotify_CloseChannel (Opnum 6)

The IRPCAsyncNotify_CloseChannel method is specified in section 3.1.4.6. Additional AsyncUI-specific server parameter details and are defined here.

**pInNotificationType:** MUST hold either notification type identifier value **AsyncPrintNotificationType_AsyncUI** or **NOTIFICATION_RELEASE**, as specified in section 2.2.1.

**pReason:** If **pInNotificationType** does hold **NOTIFICATION_RELEASE**, **pReason** MUST hold client-to-server response data conforming to the data format for **AsyncUIMessageBoxReply** or **AsyncUICustomUIReply**, as specified in sections 2.3.4 and 2.3.6, respectively. The client

sequencing rules defining the conditions under which each response format is sent can be found in section 3.6.4.

### 3.5.5 Timer Events

No timer events are required on the server beyond the timers required in the underlying Remote Procedure Call (RPC) protocol, as specified in [MS-RPCE] section 3.

### 3.5.6 Other Local Events

This protocol does not define the set of printing events that cause notification sources to trigger notifications.

## 3.6 AsyncUI Client Details

The AsyncUI  notification type MUST use the notification type identifier value **AsyncPrintNotificationType_AsyncUI**, as specified in section 2.2.1.

The AsyncUI notification type includes some notifications sent in unidirectional communication mode, and others sent in bidirectional communication mode. In bidirectional communication mode, the type of notification received by a client determines the required response type. A client of the AsyncUI notification type that uses the Print System Asynchronous Notification Protocol to process notification in bidirectional communication mode has the following state diagram when dealing with a single communication channel.

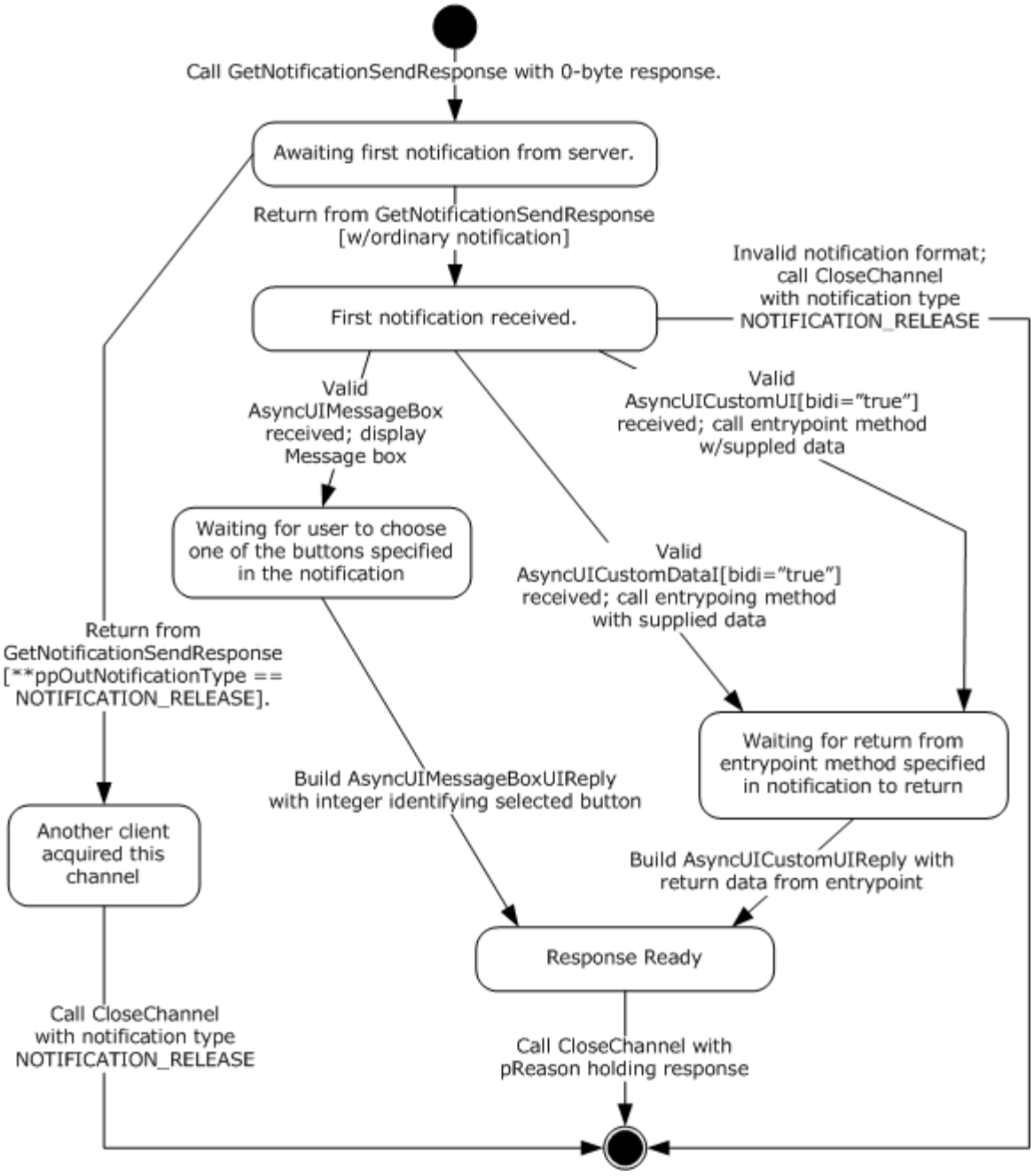


**Figure 10: Client state diagram when dealing with a single communication channel**

### 3.6.1 Abstract Data Model

No abstract data model is required.

### 3.6.2 Timers

No timer events are required on the client beyond the timers required in the underlying Remote Procedure Call (RPC) protocol, as specified in [MS-RPCE] section 3.

### 3.6.3 Initialization

A remote object client (for more information, see section 3.4), and an asynchronous notification client (for more information, see section 3.2) MUST be fully initialized on the client.

### 3.6.4 Message Processing Events and Sequencing Rules

An AsyncUI client MUST specify a notification type identifier value **AsyncPrintNotificationType_AsyncUI** (for more information, see section 2.2.1) when registering for, requesting, or responding to notifications or response data, using the methods of the Print System Asynchronous Notification Protocol.

The AsyncUI-specific syntax and behavior for each method specified in section 3.1.4 is specified in section 3.5.4.

There is no AsyncUI-specific syntax or behavior for the IRPCRemoteObject interface methods specified in section 3.3.4.

The sections which follow specify the processing of AsyncUI notifications that are delivered to a client from a printer driver, using this protocol.

#### 3.6.4.1 AsyncUIBalloon Notification

The AsyncUIBalloon notification MUST use unidirectional communication mode, and MUST be delivered by way of an output parameter of an **IRPCAsyncNotify_GetNotification** call.

Before acting on a notification, the client SHOULD<56> verify that the notification complies with all of the requirements for the AsyncUIBalloon type. If an error is detected, the client MUST NOT take any further action based on the notification data, but MUST continue with a subsequent call to **IRPCAsyncNotify_GetNotification**.

After validating the notification, an AsyncUI client MUST process the request as follows:

- The client SHOULD<57> format a display by using AsyncUI "title", "body", and "parameter" elements, as well as **iconID** and **resourceDll** attributes from the "balloonUI" element. See sections 2.3.1.3, 2.3.1.4, and 2.3.1.5 for details.
- If an "action" element is specified, the client MUST call the method identified by the **entrypoint** and **dll** attributes of that element. See section 2.3.1 for details. If the method entry point cannot be called successfully for any reason, the client MUST ignore the error and continue with a subsequent call to **IRPCAsyncNotify_GetNotification**.

#### 3.6.4.2 AsyncUIMessageBox Notification

The AsyncUIMessageBox notification MUST use bidirectional communication mode, and MUST be delivered by way of an output parameter of an IRPCAsyncNotify_GetNotificationSendResponse call. Once the notification has been processed, a client MUST NOT make an additional call to IRPCAsyncNotify_GetNotificationSendResponse by using the same **pChannel** parameter, and MUST send a response using a call to IRPCAsyncNotify_CloseChannel.

Before acting on a notification, a client SHOULD<58> verify that the notification fully complies with all of the requirements specified for AsyncUIMessageBox in section 2.3.3. If an error is detected, the client MUST NOT send any further response on the same notification channel, and MUST close the channel by calling **IRPCAsyncNotify_CloseChannel** with its **pInNotificationType** parameter holding NOTIFICATION_RELEASE.

After successfully validating the notification:

- The client SHOULD<59> format a display by using AsyncUI **title**, **body**, **parameter**, **button**, and **bitmap** elements and wait for the user to select one of the buttons.
- The client MUST:
  - Identify a selected button.
  - Construct an AsyncUIMessageBoxReply string. The **buttonID** element MUST specify the **buttonID** attribute of the **button** element that was selected. See section 2.3.3 for details.
  - Send the AsyncUIMessageBoxReply to the server in the **pReason** parameter of an **IRPCAsyncNotify_CloseChannel** call.

#### 3.6.4.3 AsyncUICustomUI Notification

The AsyncUICustomUI notification can be sent by using either unidirectional communication mode or bidirectional communication mode.

A notification sent by using unidirectional communication mode MUST be delivered by way of an output parameter of an IRPCAsyncNotify_GetNotification call.

A notification sent by using bidirectional communication mode MUST be delivered by way of an output parameter of an **IRPCAsyncNotify_GetNotificationSendResponse** call.

Once a bidirectional notification has been processed, the client MUST NOT make an additional call to **IRPCAsyncNotify_GetNotificationSendResponse** using the same **pChannel** parameter, and MUST send a response using **IRPCAsyncNotify_CloseChannel**.

Before acting on a notification, the client SHOULD<60> verify that the notification fully complies with all of the requirements specified for AsyncUICustomUI. If an error is detected, the client MUST NOT take any further action based on the notification data. If the invalid notification was sent in unidirectional communication mode, the client MUST continue with a subsequent call to **IRPCAsyncNotify_GetNotification**. If the invalid notification was sent in bidirectional communication mode, the client MUST NOT send any further response on the same notification channel, and MUST close the channel by calling **IRPCAsyncNotify_CloseChannel** with its **pInNotificationType** parameter holding **NOTIFICATION_RELEASE**.

After successfully validating the notification:

- The client MUST call the executable method identified by the **entrypoint** and **dll** attributes of the **customUI** element.
- If the **entrypoint** cannot be successfully called for any reason:
  - If the **bidi** attribute is "false", the client MUST ignore the error, and MUST continue with a subsequent call to IRPCAsyncNotify_GetNotification.
  - If the **bidi** attribute is "true":

- The client MUST stop further processing of this notification.
- The client MUST NOT send any further response on the same notification channel.
- The client MUST close the notification channel by calling **IRPCAsyncNotify_CloseChannel** with its **pInNotificationType** parameter holding **NOTIFICATION_RELEASE**.

- Otherwise, if the **entrypoint** is successful and the **bidi** attribute is "true":
  - The client MUST construct an AsyncUICustomUIReply string. The **CustomUI** element MUST contain a string that is returned by the called method.
  - The client MUST send the AsyncUICustomUIReply to the server in the **pReason** parameter of an **IRPCAsyncNotify_CloseChannel** call.

#### 3.6.4.4 AsyncUICustomData Notification

The AsyncUICustomData notification can be sent by using either unidirectional communication mode or bidirectional communication mode.

A notification sent by using unidirectional communication mode MUST be delivered by way of an output parameter of an IRPCAsyncNotify_GetNotification completion.

A notification sent by using bidirectional communication mode MUST be delivered by way of an output parameter of an **IRPCAsyncNotify_GetNotificationSendResponse** completion.

Once a bidirectional notification has been processed, the client MUST NOT make an additional call to **IRPCAsyncNotify_GetNotificationSendResponse** by using the same **pChannel** parameter, and MUST send a response by using a call to **IRPCAsyncNotify_CloseChannel**.

Before acting on a notification, the client SHOULD<61> verify that the notification fully complies with all of the requirements specified for AsyncUICustomData in section 2.3.7. If an error is detected, the client MUST NOT take any further action based on the notification data. If the invalid notification was sent in unidirectional communication mode, the client MUST continue with a subsequent call to **IRPCAsyncNotify_GetNotification**. If the invalid notification was sent in bidirectional communication mode, the client MUST NOT send any further response on the same notification channel, and MUST close the channel by calling **IRPCAsyncNotify_CloseChannel** with its **pInNotificationType** parameter holding NOTIFICATION_RELEASE.

After successfully validating the notification:

- The client MUST call the executable method identified by the **entrypoint** and **dll** attributes of the "customData" element.
- If the **entrypoint** cannot be successfully called for any reason:
  - If the **bidi** attribute is "false", the client MUST ignore the error, and MUST continue with a subsequent call to **IRPCAsyncNotify_GetNotification**.
  - If the **bidi** attribute is "true":
    - The client MUST stop further processing of this notification.
    - The client MUST NOT send any further response on the same notification channel.
    - The client MUST close the notification channel by calling **IRPCAsyncNotify_CloseChannel** with its **pInNotificationType** parameter holding NOTIFICATION_RELEASE.

- Otherwise, if the **entrypoint** is successful and the **bidi** attribute is "true":
  - The client MUST construct an AsyncUICustomUIReply string. The "CustomUI" element MUST contain a string that is returned by the called method.
  - The client MUST send the AsyncUICustomUIReply to the server in the **pReason** parameter of an **IRPCAsyncNotify_CloseChannel** call.

### 3.6.5 Timer Events

No timer events are required on the client beyond the timers required in the underlying Remote Procedure Call (RPC) protocol, as specified in [MS-RPCE] section 3.

### 3.6.6 Other Local Events

There are no AsyncUI-specific local events.

# 4 Protocol Examples

The following sections describe several operations as used in common scenarios to illustrate the function of the Print System Asynchronous Notification Protocol.

## 4.1 Unidirectional Communication Mode

This section presents an example of unidirectional communication mode, illustrating a single-server to single-client scenario. If multiple clients register with matching parameters, including notification type identifier and users privileges, each registered client would receive a copy of the notification.

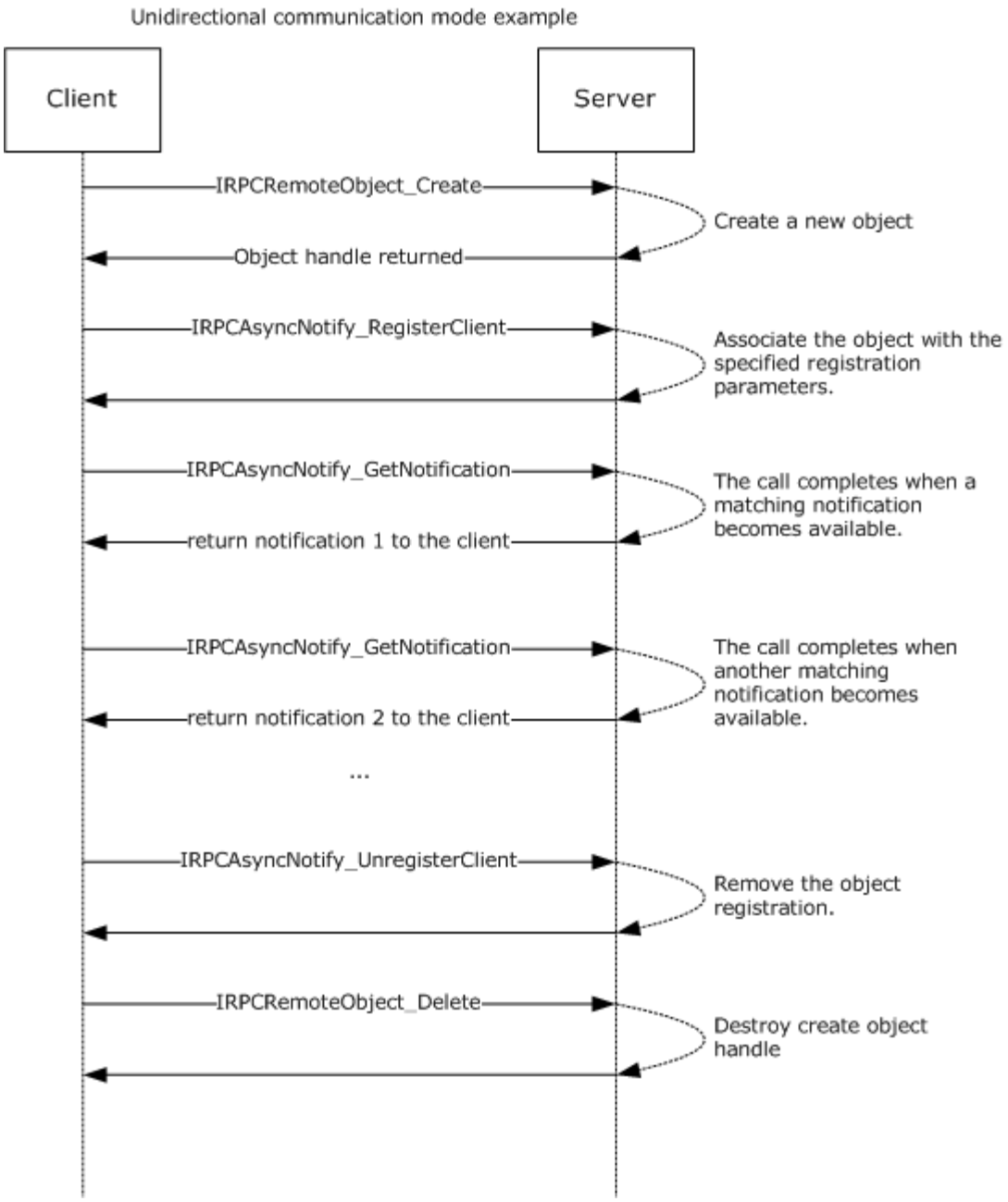


**Figure 11: Unidirectional communication mode: single-server to single-client**

## 4.2 AsyncUI Notification in Unidirectional Communication Mode

The following diagram illustrates the processing of an AsyncUI notification in unidirectional communication mode. In this example, the printer driver uses the notification type identifier value **AsyncPrintNotificationType_AsyncUI** (for more information, see section 2.2.1) to request that the client display an informative message to the user, without requesting any response to that message.

All text within the message box is identified by using references to string resource contained within a client-resident resource file.

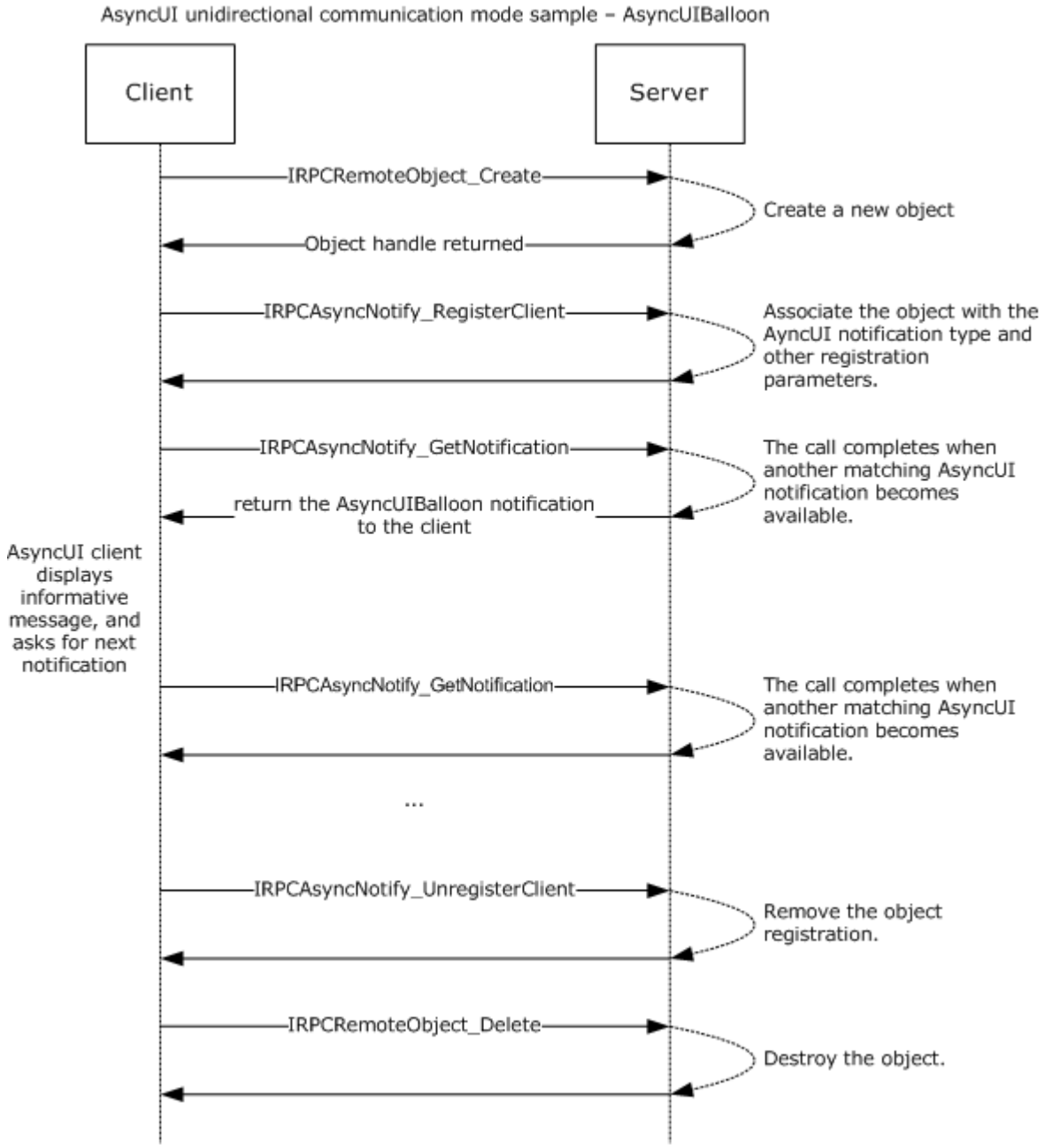


**Figure 12: Processing an AsyncUI notification in unidirectional communication mode**

Sample notification:

```
<?xml version="1.0" ?>
<asyncPrintUIRequest xmlns=
"http://schemas.microsoft.com/2003/print/asyncui/v1/request">
```

```
 <v1>
  <requestOpen>
   <balloonUI iconID="1" resourceDll="IHV.dll">
    <title stringID="1234" resourceDll="IHV.dll" />
    <body stringID="100" resourceDll="IHV.dll" />
   </balloonUI>
  </requestOpen>
 </v1>
</asyncPrintUIRequest>
```

## 4.3 Bidirectional Communication Mode

This section presents an example of bidirectional communication mode, in which only the first notification is sent to all clients registered with matching parameters, including notification type identifier and user privileges. After that, the first client to send a response to that notification acquires the notification channel. All other clients are notified that the notification channel was acquired by another client.

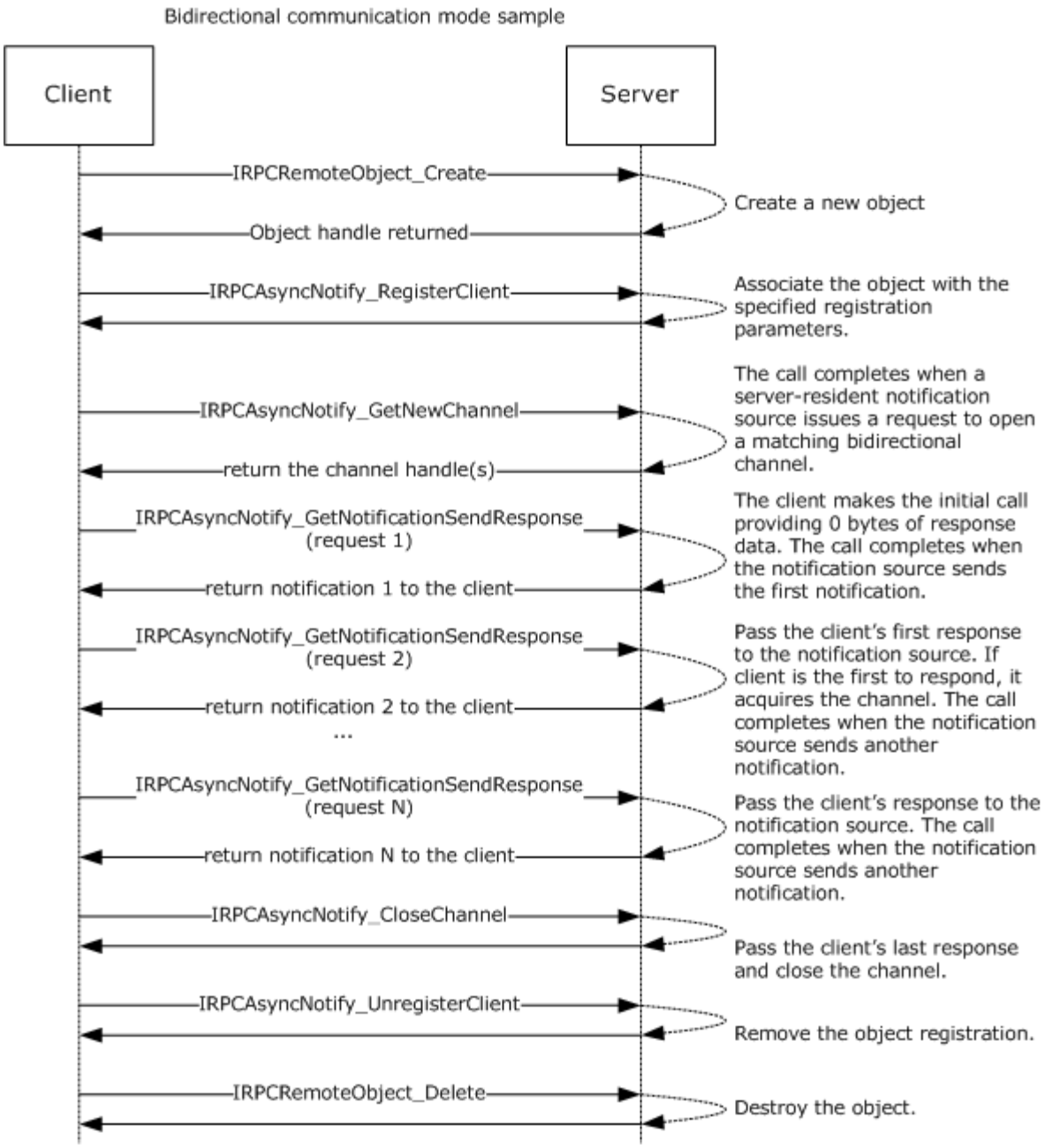


**Figure 13: Bidirectional communication mode**

## 4.4 AsyncUI Notification in Bidirectional Communication Mode

This section presents an example of processing an AsyncUI notification in bidirectional communication mode. In this example, a printer driver uses the notification type identifier value **AsyncPrintNotificationType_AsyncUI** (for more information, see section 2.2.1) to request that the client display a message box containing multiple buttons, and the client sends back a response identifying the selected button.

All text within the message box is identified by using references to string resource contained within a client-resident resource file.

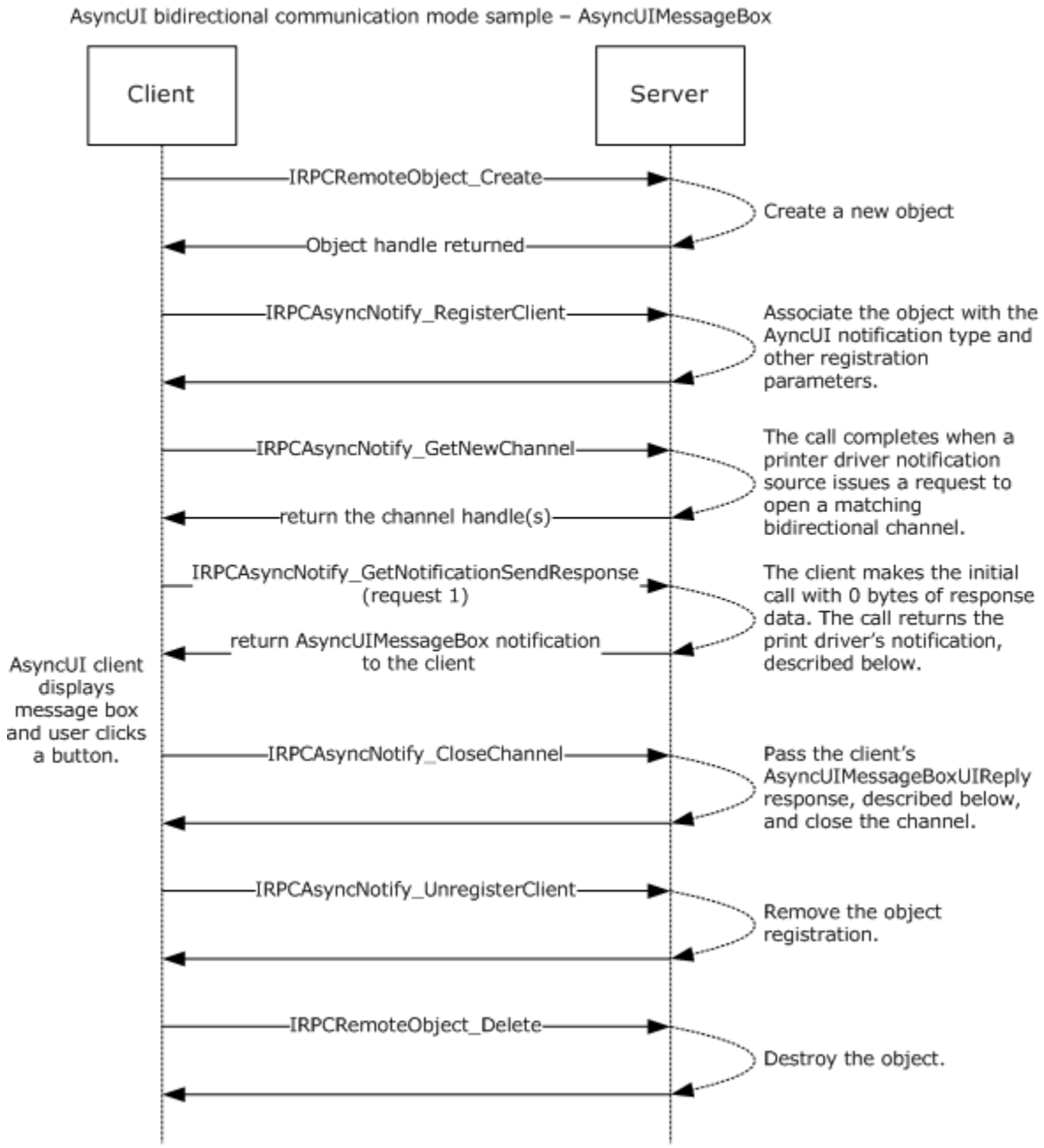


**Figure 14: Processing an AsyncUI notification in bidirectional communication mode**

Sample notification:

```
<asyncPrintUIRequest xmlns=
"http://schemas.microsoft.com/2003/print/asyncui/v1/request">
 <v1>
```

```
  <requestOpen>
   <messageBoxUI>
    <title stringID="100" resourceDll="IHV.dll" />
    <body stringID="101" resourceDll="IHV.dll" />
    <buttons>
     <button stringID="102" resourceDll="IHV.dll" buttonID="3"/>
     <button stringID="103" resourceDll="IHV.dll" buttonID="4"/>
    </buttons>
   </messageBoxUI>
  </requestOpen>
 </v1>
</asyncPrintUIRequest>
```

Sample response:

```
<asyncPrintUIResponse xmlns=
"http://schemas.microsoft.com/2003/print/asyncui/v1/response">
 <v1>
  <requestClose>
   <messageBoxUI>
    <buttonID>4</buttonID>
   </messageBoxUI >
  </requestClose>
 </v1>
</asyncPrintUIResponse>
```

# 5 Security

The following sections specify security considerations for implementers of the Print System Asynchronous Notification Protocol.

## 5.1 Security Considerations for Implementers

Security considerations for both authenticated and unauthenticated RPC used in this protocol are specified in [MS-RPCE] section 3. The client SHOULD <62> always perform authenticated RPC.

This protocol allows an individual client to request notifications meant for any and all users. See section 2.1 for relevant security specifications.

See section 1.1 for driver-file name definitions and see sections 2.3.2.1, 2.3.5.1, and 2.3.7.1 for descriptions of an AsyncUI client calling a method, identified by an **entrypoint** attribute value, within an executable driver file identified by a **dll** attribute holding a driver-file name. When calling such a method, AsyncUI clients SHOULD minimize the set of active permissions, and should guard against interpreting a driver-file name as referring to a file that is not a constituent file of a printer driver. <63>

Clients of the AsyncUI notification type SHOULD be implemented to avoid or mitigate the risk of calling arbitrary client-resident code. For information on AsyncUI elements that define **entrypoint** and **dll** attributes, see sections 2.3.2.1, 2.3.5.1, and 2.3.7.1.

## 5.2 Index of Security Parameters

This protocol defines no security parameters.

# 6 Appendix A: Full IDL

For ease of implementation, the full **IDLs** for all interfaces defined in this protocol are provided in this appendix.

## 6.1 Appendix A.1: Asynchronous Notification IDL

This appendix contains the IDL for the Asynchronous Notification interface. The file "ms-pan-remoteobject.idl" is the IDL for the remote object interface, and it is specified in in section 6.2. Some of the structures used by this protocol are defined in another document. Those structures, as specified in [MS-DTYP], are included below.

```
import "ms-pan-remoteobject.idl";

[
    uuid(0b6edbfa-4a24-4fc6-8a23-942b1eca65d1),
    version(1.0),
    pointer_default(unique)
]
interface IRPCAsyncNotify {

// [MS-PAN] enumerations
typedef [v1_enum] enum {
    kBiDirectional = 0,
    kUniDirectional = 1,
} PrintAsyncNotifyConversationStyle;

typedef [v1_enum] enum {
    kPerUser = 0,
    kAllUsers = 1,
} PrintAsyncNotifyUserFilter;

// [MS-PAN] data types
typedef GUID  PrintAsyncNotificationType;
typedef [context_handle] void* PNOTIFYOBJECT;

// [MS-PAN] methods
HRESULT
IRPCAsyncNotify_RegisterClient(
    [in] PRPCREMOTEOBJECT pRegistrationObj,
    [in,string,unique] const wchar_t* pName,
    [in] PrintAsyncNotificationType* pInNotificationType,
    [in] PrintAsyncNotifyUserFilter NotifyFilter,
    [in] PrintAsyncNotifyConversationStyle conversationStyle,
    [out, string] wchar_t** ppRmtServerReferral
);

HRESULT
IRPCAsyncNotify_UnregisterClient(
    [in] PRPCREMOTEOBJECT pRegistrationObj
);

void Opnum2NotUsedOnWire(void);

HRESULT
IRPCAsyncNotify_GetNewChannel(
```

```
    [in] PRPCREMOTEOBJECT pRemoteObj,
    [out] unsigned long* pNoOfChannels,
    [out,size_is( , *pNoOfChannels)] PNOTIFYOBJECT** ppChannelCtxt
);

HRESULT
IRPCAsyncNotify_GetNotificationSendResponse(
    [in, out] PNOTIFYOBJECT* pChannel,
    [in, unique] PrintAsyncNotificationType* pInNotificationType,
    [in] unsigned long InSize,
    [in, size_is(InSize), unique] byte* pInNotificationData,
    [out] PrintAsyncNotificationType** ppOutNotificationType,
    [out] unsigned long* pOutSize,
    [out, size_is( , *pOutSize)]  byte** ppOutNotificationData
);

HRESULT
IRPCAsyncNotify_GetNotification(
    [in] PRPCREMOTEOBJECT pRemoteObj,
    [out] PrintAsyncNotificationType** ppOutNotificationType,
    [out] unsigned long* pOutSize,
    [out, size_is( , *pOutSize)] byte** ppOutNotificationData
);

HRESULT
IRPCAsyncNotify_CloseChannel(
    [in, out] PNOTIFYOBJECT* pChannel,
    [in] PrintAsyncNotificationType* pInNotificationType,
    [in] unsigned long InSize,
    [in, size_is(InSize), unique] byte* pReason
);
}
```

## 6.2 Appendix A.2: Remote Object IDL

This appendix contains the IDL for the Remote Object interface. Some of the structures used by this protocol are defined in another document. Those structures, as specified in [MS-DTYP], are included below.

```
import "ms-dtyp.idl";

[
    uuid(ae33069b-a2a8-46ee-a235-ddfd339be281),
    version(1.0),
    pointer_default(unique)
]

interface IRPCRemoteObject
{
// [MS-PAN] data types
typedef [context_handle] void* PRPCREMOTEOBJECT;

// [MS-PAN] methods
HRESULT
```

```
IRPCRemoteObject_Create(
    [in]     handle_t           hRemoteBinding,
    [out]    PRPCREMOTEOBJECT* ppRemoteObj
);

void
IRPCRemoteObject_Delete(
    [in,out] PRPCREMOTEOBJECT* ppRemoteObj
);
}
```

# 7 Appendix B: Windows Behavior

The information in this specification is applicable to the following versions of Windows:

- Windows Server 2008
- Windows Vista

Exceptions, if any, are noted below. Unless otherwise specified, any statement of optional behavior in this specification prescribed using the terms SHOULD or SHOULD NOT implies Windows behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that Windows does not follow the prescription.

<1> Section 1.1: All Windows Vista versions: The default resource file for the Windows AsyncUI client is **prnntfy.dll**, the dynamic link library (DLL) that holds the implementation of the AsyncUI client.

The strings present in the US-English version of this resource file are listed below. Localized versions of Windows Vista or Windows Language Packs contain equivalent localized strings.

For information on the interpretation of %-character escapes used as position-parameter replacement tags, see [MSDN-FMT].

| String resource key | Corresponding text string |
|---|---|
| 100 | "..." |
| 101 | "This document was sent to the printer" |
| 102 | "Document: %1\nPrinter: %2\nTime: %3\nTotal pages: %4" |
| 103 | "Printer out of paper" |
| 104 | "Printer '%1' is out of paper." |
| 105 | "This document failed to print" |
| 106 | "Document: %1\nPrinter: %2\nTime: %3\nTotal pages: %4" |
| 107 | "Printer door open" |
| 108 | "The door on '%1' is open." |
| 109 | "Printer in an error state" |
| 110 | "'%1' is in an error state." |
| 111 | "Printer out of toner/ink" |
| 112 | "'%1' is out of toner/ink." |
| 113 | "Printer not available" |
| 114 | "'%1' is not available for printing." |
| 115 | "Printer offline" |
| 116 | "'%1' is offline." |

| String resource key | Corresponding text string |
|---|---|
| 117 | "Printer out of memory" |
| 118 | "'%1' has run out of memory." |
| 119 | "Printer output bin full" |
| 120 | "The output bin on '%1' is full." |
| 121 | "Printer paper jam" |
| 122 | "Paper is jammed in '%1'." |
| 123 | "Printer out of paper" |
| 124 | "'%1' is out of paper." |
| 125 | "Printer paper problem" |
| 126 | "'%1' has a paper problem." |
| 127 | "Printer paused" |
| 128 | "'%1' is paused." |
| 129 | "Printer needs user intervention" |
| 130 | "'%1' has a problem that requires your intervention." |
| 131 | "Printer is low on toner/ink" |
| 132 | "'%1' is low on toner/ink." |
| 600 | "OK" |
| 601 | "Cancel" |
| 1000 | "Document: %1\n" |
| 1001 | "Printer: %1\n" |
| 1002 | "Paper size: %1\n" |
| 1003 | "Ink: %1\n" |
| 1004 | "Cartridge: %1\n" |
| 1005 | "Paper jam area: %1\n" |
| 1006 | "A printer problem occurred" |
| 1007 | "Please check the printer for any problems." |
| 1008 | "Please check the printer status and settings." |
| 1009 | "Check if the printer is online and ready to print." |
| 1100 | "The printer is ready to print on the other side of the paper." |

| String resource key | Corresponding text string |
|---|---|
| 1101 | "To finish double-sided printing, remove the paper from the output tray. Re-insert the paper in the input tray, facing up." |
| 1102 | "To finish double-sided printing, remove the paper from the output tray. Re-insert the paper in the input tray, facing down." |
| 1200 | "Press the Resume button on the printer when done." |
| 1201 | "Press the Cancel button on the printer when done." |
| 1202 | "Press the OK button on the printer when done." |
| 1203 | "Press the Online button on the printer when done." |
| 1204 | "Press the Reset button on the printer when done." |
| 1300 | "The printer is offline." |
| 1301 | "Windows could not connect to your printer. Please check the connection between the computer and the printer." |
| 1302 | "The printer is not responding. Please check the connection between your computer and the printer." |
| 1400 | "Paper Jam" |
| 1401 | "Your printer has a paper jam." |
| 1402 | "Please check the printer and clear the paper jam. The printer cannot print until the paper jam is cleared." |
| 1403 | "Please clear the paper jam on the printer." |
| 1500 | "Your printer is out of paper." |
| 1501 | "Please check the printer and add more paper." |
| 1502 | "Please check the printer and add more paper in tray %1." |
| 1503 | "Please check the printer and add more %1 paper in tray %2." |
| 1600 | "The output tray on your printer is full." |
| 1601 | "Please empty the output tray on the printer." |
| 1700 | "Your printer has a paper problem" |
| 1701 | "Please check your printer for paper problems." |
| 1800 | "Your printer is out of ink" |
| 1801 | "The ink cartridge in your printer is empty." |
| 1802 | "Your printer is out of toner." |
| 1803 | "Please check the printer and add more ink." |

| String resource key | Corresponding text string |
|---|---|
| 1804 | "Please check the printer and replace the ink cartridge." |
| 1805 | "Please check the printer and add toner." |
| 2000 | "Cyan" |
| 2001 | "Magenta" |
| 2002 | "Yellow" |
| 2003 | "Black" |
| 2004 | "Light Cyan" |
| 2005 | "Light Magenta" |
| 2006 | "Red" |
| 2007 | "Green" |
| 2008 | "Blue" |
| 2009 | "Gloss optimizer" |
| 2010 | "Photo Black" |
| 2011 | "Matte Black" |
| 2012 | "Photo Cyan" |
| 2013 | "Photo Magenta" |
| 2014 | "Light Black" |
| 2015 | "Ink optimizer" |
| 2016 | "Blue photo" |
| 2017 | "Gray photo" |
| 2018 | "Tricolor photo" |
| 2100 | "Cyan cartridge" |
| 2101 | "Magenta cartridge" |
| 2102 | "Black cartridge" |
| 2103 | "CMYK cartridge" |
| 2104 | "Gray cartridge" |
| 2105 | "Color cartridge" |
| 2106 | "Photo cartridge" |
| 2200 | "A door on your printer is open." |

| String resource key | Corresponding text string |
|---|---|
| 2201 | "A cover on your printer is open." |
| 2202 | "Please check the printer and close any open doors. The printer cannot print while a door is open." |
| 2203 | "Please check the printer and close any open covers. The printer cannot print while a cover is open." |
| 2300 | "Your printer is not printing" |
| 2301 | "Please check your printer" |
| 2302 | "Your printer is out of memory" |
| 2303 | "Your document might not print correctly. Please see online help." |
| 2400 | "Your printer is low on ink" |
| 2401 | "The ink cartridge in your printer is almost empty." |
| 2402 | "Your printer is low on toner" |
| 2403 | "Please check the printer and add more ink when needed." |
| 2404 | "Please check the printer and replace the ink cartridge when needed." |
| 2405 | "Please check the printer and add toner when needed." |
| 2500 | "The ink system in your printer is not working" |
| 2501 | "The ink cartridge in your printer is not working" |
| 2502 | "The toner system in your printer is not working" |
| 2503 | "Please check the ink system in your printer." |
| 2504 | "Please check the ink cartridge in your printer." |
| 2505 | "Please check the toner system in your printer." |
| 2506 | "Please check that the ink cartridge was installed correctly in the printer." |
| 2600 | "Printer has been paused" |
| 2601 | "'%1' cannot print, because it has been put into a paused state at the device." |
| 2602 | "'%1' cannot print, because it has been put into an offline state at the device." |
| 2700 | "Your document has been printed." |
| 2701 | "Your document is in the output tray." |
| 2702 | "%1!d! document(s) pending for %2" |
| 2703 | "<unknown>" |

<2> Section 1.1: All Windows versions: The Windows AsyncUI client interprets a driver-file name relative to the directory into which a printer driver was installed (for example, %SYSTEMROOT%\system32\spool\drivers\w32x86\3 for an x86 based Version 3 driver on Windows Vista).

<3> Section 1.1: All Windows versions: The string resources of the default resource file are localized. The string resources of inbox printer drivers included with Windows are localized. The author of any other printer driver determines whether or not to localize that driver's string resources.

<4> Section 1.1: All Windows versions: The AsyncUI client uses the user locale to determine the language of retrieved strings.

<5> Section 1.8: All Windows versions: Aside from the HRESULT values explicitly defined in this specification, Windows uses return values as specified in [MS-ERREF].

<6> Section 2.2.1: All Windows versions: The specific **GUIDs** in the table in section 2.2.1 were generated by using the Windows "guidgen" application.

<7> Section 2.2.2: All Windows versions: Based on the Windows Security Model, the Print System Asynchronous Notification Protocol will only allow principals with **SERVER_ALL_ACCESS**, **PRINTER_ALL_ACCESS** or privileges to issue filters that will enable the caller to get messages intended for all users. **PRINTER_ALL_ACCESS** and **SERVER_ALL_ACCESS** privileges are specified in [MS-RPRN] section **2.2.3.4**.

<8> Section 2.3.1.3: All Windows versions: The string resource identified by the **title** element is displayed as the title of a dialog or balloon notification. For more information, see [MSDN-FMT].

<9> Section 2.3.1.3: All Windows versions: Implementations interpret tags such as %1 or %2 as parameter replacement markers.

<10> Section 2.3.1.4: All Windows versions: Individual device drivers build their own notification strings, and each can make its own choice to use one or more **body** elements.

<11> Section 2.3.1.4: All Windows versions: The Windows system interprets tags such as %1 or %2 as parameter replacement markers. For more information, see [MSDN-FMT].

<12> Section 2.3.1.5: All Windows versions: If **stringID** is present but the client is unable to load a corresponding string resource, the nested text is used.

<13> Section 2.3.2.1: All Windows versions: The AsyncUI client treats as an error a driver-file name containing any of the following [UNICODE] characters:

\ (**Unicode** character U+005C)

/ (Unicode character U+002F)

? (Unicode character U+003F)

* (Unicode character U+002A)

< (Unicode character U+003C)

> (Unicode character U+003E)

" (Unicode character U+0022)

| (Unicode character U+007C)

: (Unicode character U+003A)

<14> Section 2.3.2.1: All Windows versions: The function signature of the method specified by the **entrypoint** attribute is:

```
void entrypoint(void * data);
```

**data:** MUST be the data from the **action** element.

<15> Section 2.3.2.1: All Windows versions: The element's content is passed in UTF-16LE-encoded form, as specified in [RFC2781] section 4.2.

<16> Section 2.3.2.2: All Windows versions: If an **iconID** attribute is not provided, implementations display a generic printer icon in the balloon message.

<17> Section 2.3.2.2: If the **resourceDll** attribute is not specified, a generic printer icon is used in the balloon message.

<18> Section 2.3.3.1: All Windows versions: "IDOK" is specified for an "OK" or "YES" button event, and "IDCANCEL" is specified for a "CANCEL" or "NO" button event.

<19> Section 2.3.3.1: All Windows versions: The IDOK button has initial input focus within the dialog. If the user presses the keyboard ENTER key when no other button in the dialog has focus, the system behaves as though this button were clicked.

<20> Section 2.3.3.1: All Windows versions: If the user presses the keyboard ESC key in the dialog, the system behaves as though this button were clicked. Similarly, if the user dismisses the dialog without pressing a button, the system behaves as if this button were clicked.

<21> Section 2.3.3.4: The AsyncUI client on Windows displays a message box constructed using title, bitmap, body, and button resources.

<22> Section 2.3.5.1: All Windows versions: The AsyncUI client treats as an error a driver-file name containing any of the following [UNICODE] characters:

\ (Unicode character U+005C)

/ (Unicode character U+002F)

? (Unicode character U+003F)

* (Unicode character U+002A)

< (Unicode character U+003C)

> (Unicode character U+003E)

" (Unicode character U+0022)

| (Unicode character U+007C)

: (Unicode character U+003A)

<23> Section 2.3.5.1: All Windows versions: The function signature of the method specified by the **entrypoint** attribute varies depending on the **bidi** attribute value.

| bidi value | Method prototype |
|---|---|
| "true" | void * **entrypoint**(void * **data:**) |
| "false" | void **entrypoint**(void * **data:**) |

**data:** The data from the **customUI** element.

<24> Section 2.3.5.1: All Windows versions: The element's content is passed in UTF-16LE-encoded form, as specified in [RFC2781] section 4.2.

<25> Section 2.3.5.1: All Windows versions: The value returned by the **entrypoint** method is interpreted as a UTF-16LE-encoded string, as specified in [RFC2781] section 4.2.

<26> Section 2.3.7.1: All Windows versions: The AsyncUI client treats as an error a driver-file name containing any of the following [UNICODE] characters:

\ (Unicode character U+005C)

/ (Unicode character U+002F)

? (Unicode character U+003F)

* (Unicode character U+002A)

< (Unicode character U+003C)

> (Unicode character U+003E)

" (Unicode character U+0022)

| (Unicode character U+007C)

: (Unicode character U+003A)

<27> Section 2.3.7.1: All Windows versions: The function signature of the method specified by the **entrypoint** attribute depends on the **bidi** attribute value, as shown below.

| bidi **Value** | **Method Prototype** |
|---|---|
| "true" | void * **entrypoint**(void * data) |
| "false" | void **entrypoint**(void * data) |

**data:** The data from the **customData** element.

<28> Section 2.3.7.1: All Windows versions: The value returned by the **entrypoint** method is interpreted as a UTF-16LE-encoded string, as specified in [RFC2781] section 4.2.

<29> Section 3.1.4: Gaps in the opnum numbering sequence apply to Windows as follows.

| Opnum | Description |
|---|---|
| 2 | Deprecated and not defined. It is never used. |

<30> Section 3.1.4.1: All Windows versions: The server always returns a null value in this parameter.

<31> Section 3.1.4.1: All Windows versions: The server does not detect that the remote object represented by **pRegistrationObj** was already associated with a set of registration parameters from a prior call to **IRPCAsyncNotify_RegisterClient**, and it returns an HRESULT value of 0x00000000.

<32> Section 3.1.4.1: All Windows versions: The server returns an HRESULT value of:

- 0x80070005 to indicate that the client does not have authorization to register for notifications with the set of parameters specified in this call.
- 0x8007000E to indicate that the server does not have enough memory resources to complete the call successfully.
- 0x80070015 to indicate that the server cannot accept new registrations, because the number of registered remote objects has reached the server's maximum.
- 0x80070057 to indicate that the operation failed due to one or more parameters being invalid.
- 0x8007007B to indicate that if the value of the **pName** parameter is not null, the value does not identify a printer on the print server.

<33> Section 3.1.4.1: All Windows versions: A client never re-registers an unregistered PRPCREMOTEOBJECT; it always creates a new remote object for each registration.

<34> Section 3.1.4.2: All Windows versions: The server returns an HRESULT value of:

- 0x80070057 to indicate that the remote object represented by **pRegistrationObj** was unregistered by a prior call to **IRPCAsyncNotify_UnregisterClient**.
- 0x80070057 to indicate that the remote object is in an invalid state due to a prior failed call to **IRPCAsyncNotify_RegisterClient**.

<35> Section 3.1.4.3: All Windows versions: The server returns all not-yet-acquired bidirectional channels in response to an **IRPCAsyncNotify_GetNewChannel** call. This is true whether the channels were created before or after client registration or a call to **IRPCAsyncNotify_GetNewChannel**.

<36> Section 3.1.4.3: All Windows versions: Clients implement auto-retry logic with attempts spaced at one-minute intervals. Retry continues until an application deregisters the given remote object. For AsyncUI notification, a client will continue the retries indefinitely.

<37> Section 3.1.4.3: All Windows versions: The server does not return an error and blocks the call indefinitely.

<38> Section 3.1.4.3: All Windows versions: The server propagates the values 0x8004000C, 0x8004000E, and 0x8008071A as exception codes, not as return values.

<39> Section 3.1.4.3: All Windows versions: Clients have dedicated threads that make calls to **IRPCAsyncNotify_GetNewChannel**, as soon as calls to **IRPCAsyncNotify_RegisterClient**, or prior calls to **IRPCAsyncNotify_GetNewChannel**, successfully return.

<40> Section 3.1.4.3: All Windows versions: Servers return all not-yet-acquired bidirectional channels in response to **IRPCAsyncNotify_GetNewChannel** calls. This applies to channels whether they were created before or after client registration or a call to **IRPCAsyncNotify_GetNewChannel**.

<41> Section 3.1.4.4: All Windows versions: The server returns an error on all calls that provide values larger than 0x00A00000.

<42> Section 3.1.4.4: All Windows versions: Client implementations do not restrict this value.

<43> Section 3.1.4.4: All Windows versions: The server returns an HRESULT value of:

- 0x80040008 to indicate that the notification channel represented by **pChannel** was closed due to a prior call to **IRPCAsyncNotify_CloseChannel**.
- 0x8004000C to indicate that a prior call to **IRPCAsyncNotify_GetNotificationSendResponse** using the same **pChannel** parameter MUST complete before the server can process incoming calls of **IRPCAsyncNotify_GetNotificationSendResponse**.
- 0x80040012 to indicate that the size of the client-to-server notification exceeded the maximum size of 0x00A00000.
- 0x80040014 to indicate that the notification type identifier is different from that of the notification channel's notification type.

<44> Section 3.1.4.5: All Windows versions: Client implementations do not restrict this value.

<45> Section 3.1.4.5: All Windows versions: Clients implement auto-retry logic with attempts spaced at one-minute intervals. Retry continues until an application deregisters the given remote object. For AsyncUI notifications, a client will continue the retries indefinitely.

<46> Section 3.1.4.5: All Windows versions: The server does not return an error and blocks the call indefinitely.

<47> Section 3.1.4.5: All Windows versions: The server does not return an error and blocks the call indefinitely.

<48> Section 3.1.4.5: All Windows versions: The server propagates the values 0x8004000C, 0x8004000E and 0x8008071A as exception codes, not as return values.

<49> Section 3.1.4.5: All Windows versions: The server implementation does not buffer any unidirectional notifications in the absence of corresponding registered clients. The server does buffer up to 100 unidirectional notifications for each registered client that does not have pending **IRPCAsyncNotify_GetNotification** calls.

<50> Section 3.1.4.5: All Windows versions: Clients have dedicated threads that make calls to **IRPCAsyncNotify_GetNotification** as soon as calls to **IRPCAsyncNotify_RegisterClient**, or prior calls to **IRPCAsyncNotify_GetNotification**, successfully return.

<51> Section 3.1.4.6: All Windows versions: The server implementation returns an error on all calls that provide values larger than 0x00A00000.

<52> Section 3.1.4.6: All Windows versions: The server returns an HRESULT value of:

- 0x00040008 to indicate that the notification channel represented by **pChannel** was closed due to a prior call to **IRPCAsyncNotify_CloseChannel**.
- 0x00040010 to indicate that the notification channel was acquired by a different client.
- 0x80040012 to indicate that the size of the client-to-server notification exceeded the maximum size of 0x00A00000.
- 0x80040014 to indicate that the notification type identifier is different from that of the notification channel's notification type.

<53> Section 3.2.2: All Windows versions: Although timers are not required, client implementations automatically attempt to reconnect to the server in case of a failure every minute, until the connection is re-established or the client unregisters the remote object.

<54> Section 3.4.2: All Windows versions: Although timers are not required, client implementations automatically attempt to reconnect to the server in case of a failure, every minute, until the connection is re-established or the client unregisters the remote object.

<55> Section 3.5.4: Gaps in the opnum numbering sequence apply to Windows as follows.

| Opnum | Description |
|---|---|
| 2 | Deprecated and not defined. It is never used. |

<56> Section 3.6.4.1: All Windows versions: Clients do not fully validate notification conformance with all requirements. In particular:

- Clients use case-insensitive comparisons when validating XML-element names.
- Clients use case-insensitive comparisons when validating and processing the values of the **bidi** and **buttonID** attributes.
- Although XML schemas specify an ordering for sibling elements, clients accept those elements in any order.
- Clients ignore XML attributes with unrecognized names.
- Although this protocol specifies integer string encodings:
  - Clients accept any string.
  - If leading, contiguous non-white-space characters of the string can be decoded as an integer, clients accept the integer and discard remaining characters.
  - If no leading characters can be decoded as an integer, clients treat the string as if it held the value "0".
- Clients accept any string for the value of the **bidi** attribute, and treat any value other than "true" as if it were "false".
- Clients accept a "messageBoxUI" or "balloonUI" element that lacks the required "body" element.

<57> Section 3.6.4.1: All Windows versions: Clients format balloon-style dialog boxes using AsyncUI elements and attributes.

<58> Section 3.6.4.2: All Windows versions: Clients do not fully validate notification conformance with all requirements. See the Windows behavior note in section 3.6.4.1 regarding notification validation.

<59> Section 3.6.4.2: All Windows versions: Clients format dialog boxes by using AsyncUI elements and attributes.

<60> Section 3.6.4.3: All Windows versions: Clients do not fully validate notification conformance with all requirements. See the Windows behavior note in section 3.6.4.1 regarding notification validation.

<61> Section 3.6.4.4: All Windows versions: Clients do not fully validate notification conformance with all requirements. See the Windows behavior note in section 3.6.4.1 regarding notification validation.

<62> Section 5.1: All Windows versions: The Print System Asynchronous Notification Protocol server implementation follows a security model in which the print server, **print queues**, and **print jobs** are securable resources. Each of these resources has a **security descriptor** associated with it. The security descriptor contains the security information associated with a resource on the print server. The print server checks the client's access to resources by comparing the security information associated with the caller against the resource's security descriptor.

Each RPC client has associated with it an access token, which contains the **security identifier (SID)** of the user making the RPC call. The security descriptor identifies the printing resource's owner and contains a **discretionary access control list (DACL)**. The DACL contains **access control entries (ACEs)**, which specify the SID of a user or group of users, as well as whether access rights are to be allowed, denied, or audited. For resources on a print server, the ACEs specify operations including printing, managing printers, and managing documents in a print queue.

The security descriptor associated with the print server controls the client's registration and creation of remote objects and notification channels, as well as the outcome of subsequent operations on the remote objects and notification channels.

When the client calls **IRPCAsyncNotify_RegisterClient**, the client must specify the desired user filter with the NotifyFilter parameter that will be applied to subsequent operations, for which the given **PRPCREMOTEOBJECT** handle is being registered. If the caller has the required permissions, the registration completes successfully, and the registered **PRPCREMOTEOBJECT** handle can be used for subsequent calls (for example, **IRPCAsyncNotify_GetNewChannel**, **IRPCAsyncNotify_GetNotification**).

<63> Section 5.1: All Windows versions: The AsyncUI client treats as an error a driver-file name containing any of the following [UNICODE] characters:

\ (Unicode character U+005C)

/ (Unicode character U+002F)

? (Unicode character U+003F)

* (Unicode character U+002A)

< (Unicode character U+003C)

> (Unicode character U+003E)

" (Unicode character U+0022)

| (Unicode character U+007C)

: (Unicode character U+003A)

# 8 Index

## L



## M



## N



## O



## P



## R



## S



## T



## U



## V

## W