MFi Accessory Firmware Specification

Release R46

2012-09-12

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Chapter 2 **2. Protocol Core 87**

Chapter 3 **3. The General Lingo 119**

Figures, Tables, and Listings

Chapter 4

Appendix A A. iTunes Tagging Accessory Design 467

0. Introduction

This document specifies the firmware interfaces to the Apple iPod, iPhone, and iPad that are available to accessories. The models covered by this specification are shown in Table I-1 (page 32). This document does not apply to the first- and second-generation iPods, nor to the iPod shuffle.

IMPORTANT: This document uses the term "Apple device" to refer generically to iPods, iPhones, and iPads, all of which support the iPod Accessory Protocol (iAP) interface. Among these products, those that also run iOS (Apple's mobile operating system) are referred to as "iOS devices." Specifications in this document that are designated for iOS devices apply only to those products. Specifications designated for iPods apply only to Apple devices that are not iOS devices.

To determine if a command or feature is supported in a particular Apple device model or firmware release, see Table 1-5 (page 44), Table 1-6 (page 45), and Table 1-7 (page 46).

Note: All accessories for the iPhone must meet additional electrical and RF design requirements. These requirements are described in the iPhone license agreement and in Apple's *MFi Accessory Testing Specification*.

The minimum iPod firmware and iOS versions supported by this specification are:

- iPod firmware version 3.0 on the fourth-generation (4G) iPod.
- iPod firmware version 1.0 on the iPod mini, fourth-generation iPod (color display), iPod nano, fifth-generation iPod (5G), second-generation iPod nano, iPod classic, and iPod 3G nano.
- iPod firmware version 1.0.2 on the 4G nano.
- iOS version 1.1 on the iPhone, iPhone 3G, and iPod touch.
- iOS version 2.1.1 on the 2G iPod touch.
- iOS version 3.0.0 on the iPhone 3GS.
- iOS version 3.2.0 on the iPad.
- iOS version 4.0.0 on the iPhone 4.

Note: Supporting the 3G iPod requires special design considerations. See "The 3G iPod" (page 527) and "Interfacing With the 3G iPod" in *MFi Accessory Hardware Specification*.

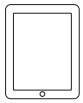
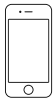
Table I-1 Apple device models

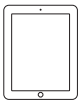
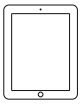
iPod mini		
	*Product name:* iPod mini	*Shipped:* 01/2004
	Compatibility icon:  iPod mini 4GB 6GB	
	Connectivity: Dock connector, headphone jack.	
4G iPod		
	*Product names:* iPod (4th generation), iPod photo, iPod photo (2nd generation), iPod 4th generation (color display)	*Shipped:* 07/2004, 10/2004, 02/2005, 06/2005
	Compatibility icons:  iPod 4th generation 20GB  iPod 4th generation 40GB  iPod 4th generation (color display) 20GB 30GB  iPod 4th generation (color display) 40GB 60GB	
	*Connectivity:* Dock connector, headphone jack.	
iPod nano		
	*Product name:* iPod nano	*Shipped:* 09/2005
	Compatibility icon:  iPod nano 1st generation 1GB 2GB 4GB	
	Connectivity: Dock connector, headphone jack.	

5G iPod		
	*Product name:* iPod with video	*Shipped:* 10/2005
	Compatibility icons: **iPod** 5th generation (video) 30GB **iPod** 5th generation (video) 60GB 80GB	
	Connectivity: Dock connector, headphone jack.	
2G nano		
	*Product name:* iPod nano (2nd generation)	*Shipped:* 09/2006
	Compatibility icon: **iPod nano** 2nd generation (aluminum) 2GB 4GB 8GB	
	Connectivity: Dock connector, headphone jack.	
iPod classic		
	*Product names:* iPod classic	*Shipped:* 09/2007
	Compatibility icon: **iPod classic** 80GB **iPod classic** 160GB (2007)	
	Connectivity: Dock connector, headphone jack.	
120 GB classic, 160 GB classic		
	*Product names:* iPod classic (120GB), iPod classic (160GB)	*Shipped:* 09/2008, 9/2009
	Compatibility icon: **iPod classic** 160GB (2009)	
	Connectivity: Dock connector, microphone/headphone jack.	

3G nano		
	*Product name:* iPod nano (3rd generation)	*Shipped:* 09/2007
	Compatibility icon: **iPod nano** 3rd generation (video) 4GB 8GB	
	Connectivity: Dock connector, headphone jack.	
4G nano		
	*Product name:* iPod nano (4th generation)	*Shipped:* 09/2008
	Compatibility icon: **iPod nano** 4th generation (video) 8GB 16GB	
	Connectivity: Dock connector, microphone/headphone jack.	
iPhone		
	*Product name:* iPhone	*Shipped:* 06/2007
	Compatibility icon: **iPhone** 4GB 8GB 16GB	
	Connectivity: GSM, Wi-Fi, Bluetooth, multi-touch display, dock connector, microphone/headphone jack. *Sensors:* Still camera.	
iPod touch		
	*Product names:* iPod touch	*Shipped:* 09/2007
	Compatibility icon: **iPod touch** 1st generation 8GB 16GB 32GB	
	Connectivity: Wi-Fi, multi-touch display, dock connector, headphone jack.	

iPhone 3G		
	*Product name:* iPhone 3G	*Shipped:* 06/2008
	Compatibility icon:  iPhone 3G 8GB 16GB	
	Connectivity: GSM, Wi-Fi, Bluetooth, multi-touch display, dock connector, microphone/headphone jack. *Sensors:* Still camera, assisted GPS.	
2G touch		
	*Product names:* iPod touch (2nd generation)	*Shipped:* 09/2008
	Compatibility icon:  iPod touch 2nd generation 8GB 16GB 32GB	
	Connectivity: Wi-Fi, Bluetooth, multi-touch display, dock connector, microphone/headphone jack.	
iPhone 3GS		
	*Product name:* iPhone 3GS	*Shipped:* 6/2009, 6/2010
	Compatibility icon:  iPhone 3GS 8GB 16GB 32GB	
	Connectivity: GSM, Wi-Fi, Bluetooth, multi-touch display, dock connector, microphone/headphone jack. *Sensors:* Still/video camera, assisted GPS, digital compass.	
5G nano		
	*Product name:* iPod nano (5th generation)	*Shipped:* 09/2009
	Compatibility icon:  iPod nano 5th generation (video camera) 8GB 16GB	
	Connectivity: Dock connector, microphone/headphone jack.	

<table>
<tr><th colspan="3">3G touch</th></tr>
<tr><td rowspan="3">  </td><td>Product names: iPod touch (3rd generation)</td><td>Shipped: 09/2009</td></tr>
<tr><td colspan="2">Compatibility icon:
 
iPod touch
3rd generation
32GB 64GB</td></tr>
<tr><td colspan="2">Connectivity: Wi-Fi, Bluetooth, multi-touch display, dock connector, microphone/headphone jack.</td></tr>
<tr><th colspan="3">iPad</th></tr>
<tr><td rowspan="3">  </td><td>Product names: iPad (Wi-Fi), iPad (Wi-Fi + 3G)</td><td>Shipped: 04/2010</td></tr>
<tr><td colspan="2">Compatibility icon:
 
iPad
16GB 32GB 64GB</td></tr>
<tr><td colspan="2">Connectivity: GSM (iPad 3G), Wi-Fi, Bluetooth, multi-touch display, dock connector, microphone/headphone jack. Sensors: Assisted GPS, digital compass, accelerometer, ambient light sensor.</td></tr>
<tr><th colspan="3">iPhone 4</th></tr>
<tr><td rowspan="3">  </td><td>Product names: iPhone 4 (GSM model), iPhone 4 (CDMA model), iPhone 4S</td><td>Shipped: 06/2010, 02/2011, 10/2011</td></tr>
<tr><td colspan="2">Compatibility icons:
 
iPhone 4
16GB 32GB
 
iPhone 4S
16GB 32GB 64GB</td></tr>
<tr><td colspan="2">Connectivity: CDMA or GSM (both in iPhone 4S), Wi-Fi, Bluetooth, multi-touch display, dock connector, microphone/headphone jack. Sensors: Two still/video cameras, assisted GPS, digital compass, 3-axis gyroscope.</td></tr>
<tr><th colspan="3">6G nano</th></tr>
<tr><td rowspan="3">  </td><td>Product name: iPod nano (6th generation)</td><td>Shipped: 09/2010</td></tr>
<tr><td colspan="2">Compatibility icon:
 
iPod nano
6th generation
8GB 16GB</td></tr>
<tr><td colspan="2">Connectivity: Multi-touch display, dock connector, microphone/headphone jack.</td></tr>
</table>

4G touch		
	*Product names:* iPod touch (4th generation)	*Shipped:* 09/2010
	Compatibility icon:  **iPod touch** 4th generation 8GB 32GB 64GB	
	Connectivity: Wi-Fi, Bluetooth, multi-touch display, dock connector, microphone/headphone jack. *Sensors:* Two still/video cameras, 3-axis gyroscope.	
iPad 2		
	*Product names:* iPad 2 (Wi-Fi), iPad 2 (Wi-Fi + 3G)	*Shipped:* 03/2011
	Compatibility icon:  **iPad 2** 16GB 32GB 64GB	
	Connectivity: CDMA or GSM (iPad 2 (Wi-Fi + 3G)), Wi-Fi, Bluetooth, multi-touch display, dock connector, microphone/headphone jack. *Sensors:* Still camera, still/video camera, assisted GPS, digital compass, accelerometer, 3-axis gyroscope, ambient light sensor.	
iPad (3rd generation)		
	*Product names:* iPad with Wi-Fi (3rd generation), iPad with Wi-Fi + 4G (3rd generation)	*Shipped:* 03/2012
	Compatibility icon:  **iPad** (3rd generation) 16GB 32GB 64GB	
	Connectivity: CDMA or GSM (iPad with Wi-Fi + 4G (3rd generation)), Wi-Fi, Bluetooth, multi-touch display, dock connector, microphone/headphone jack. *Sensors:* Still camera, still/video camera, assisted GPS, digital compass, accelerometer, 3-axis gyroscope, ambient light sensor.	

Organization of This Document

The specifications in this document are arranged in five chapters:

- "Protocol Features and Availability" (page 41) gives an overview of the iAP lingoes and their availability.
- "Protocol Core" (page 87) specifies the various functions of the iPod Accessory Protocol.
- "The General Lingo" (page 119) describes the General Lingo, which all accessories must support.

- "Additional Lingoes" (page 211) describes the various task-specific lingoes that are part of the iAP and specifies their commands.
- "Extended Interface Mode" (page 397) describes iAP Lingo 0x04, which allows the Apple device user interface to be translated to other environments.

Several appendixes provide additional information for both hardware and software designers:

- "iTunes Tagging Accessory Design" (page 467) describes the requirements for broadcast reception accessories to support iTunes tagging.
- "Nike + iPod Cardio Accessory Design" (page 501) describes the requirements for using the Sports and Storage lingoes in Nike + iPod cardio accessories.
- "Historical Information" (page 523) provides specifications for past Apple devices and iAP protocols.

At the end of this document are a glossary of terms and a document revision history.

0.1 General Specification Terms

Parts of this document contain specification requirements that are incorporated by reference into legal agreements between Apple Inc. and its licensees. The use of the words "must," "should," and "may" in these specifications have the following meanings:

- "Must" means that the specification is an absolute requirement.
- "Must not" means that the specification is an absolute prohibition.
- "Should" means that there may be valid reasons in particular circumstances to ignore the specification, but their full implications must be understood and carefully weighed before choosing to do so.
- "Should not" means that there may be valid reasons in particular circumstances that make the specified action or feature acceptable, but their full implications must be understood and carefully weighed before choosing to include it.
- "May" means that the indicated action or feature does not contravene this specification.

0.2 Special Terminology

Certain terms in this document have the following specific meanings:

- When a data field is marked "Reserved," accessories writing to it must set it to 0 (unless otherwise noted) and accessories reading it must ignore its value.
- "Deprecated" marks an earlier technology that is no longer permitted in new accessory designs.
- "USB Host Mode" is an operating mode in which an Apple device is a USB host and its attached accessory acts as a USB device.
- "USB device mode" is an operating mode in which an accessory is a USB host and its attached Apple device acts as a USB device.

See Also

For further information, refer to the latest revisions of these additional documents:

- *IEEE 1394a Specification*
- *USB 2.0 High Speed Specification*
- *USB Device Class Definition for Audio Devices*
- *USB Device Class Definition for Human Interface Devices (HID)*
- *United States RBDS Standard, NRSC-4-A*

0. Introduction

1. Protocol Features and Availability

Table 1-1 (page 41) lists the current firmware and OS versions for each iPod, iPhone, and iPad model.

Table 1-1 Current firmware and OS versions in Apple devices

Model	Firmware version	Revision date	OS version	Revision date
iPod mini	1.4.1	01/2006		
4G iPod	3.1.1	01/2006		
4G iPod (color display)	1.2.1	01/2006		
1G nano	1.3.1	03/2007		
iPod 5G	1.3	03/2008		
2G nano	1.1.3	05/2007		
iPod classic	1.1.2	05/2008		
3G nano	1.1.3	07/2008		
iPod touch			iOS 3.1.3	02/2010
iPhone			iOS 3.1.3	02/2010
iPhone 3G			iOS 4.2.1	11/2010
4G nano	1.0.4	08/2009		
120 GB classic	2.0.1	01/2009		
2G touch			iOS 4.2.1	11/2010
iPhone 3GS			iOS 6.0	09/2012
160 GB classic	2.0.4	12/2009		
5G nano	1.0.2	11/2009		
iPad			iOS 5.1	03/2012
iPad 3G			iOS 5.1	03/2012
3G touch			iOS 5.1	03/2012
6G nano	1.2	10/2011		

Model	Firmware version	Revision date	OS version	Revision date
4G touch			iOS 6.0	03/2012
iPhone 4 (all models)			iOS 6.0	09/2012
iPad 2 (Wi-Fi)			iOS 6.0	09/2012
iPad 2 (Wi-Fi + 3G)			iOS 6.0	09/2012
iPad (3rd generation)			iOS 6.0	09/2012

Table 1-2 (page 42), Table 1-3 (page 42), and Table 1-4 (page 43) list the lingo protocol versions for each Apple device's most recent firmware and OS version. In these tables, "NL" indicates that the given lingo was not implemented in the specified model loaded with the specified firmware. "NV" indicates that although the lingo was implemented, its protocol version could not be read from the iPod.

Table 1-2 Most recent lingo versions for Apple devices, Table 1

Models	mini	4G	photo; 4G (color display)	1G nano	5G	2G nano	classic	3G nano	iPod touch
General 0x00	1.02	1.02	1.02	1.05	1.06	1.06	1.07	1.07	1.09
Simple Remote 0x02	1.00	1.00	1.00	1.02	1.02	1.02	1.02	1.02	1.02
Display Remote 0x03	1.01	1.01	1.01	1.05	1.05	1.04	1.05	1.05	1.05
Extended Interface 0x04	1.05	1.05	1.09	1.11	1.12	1.10	1.13	1.13	1.12
USB Host Mode 0x06	NL	NL	NL	NL	NL	NL	NL	NL	NL
RF Tuner 0x07	NL	NL	NL	1.00	1.00	1.00	1.00	1.00	NL
Accessory Equalizer 0x08	NL	NL	NL	1.00	1.00	1.00	1.00	1.00	1.00
Sports 0x09	NL	NL	NL	NL	NL	NL	NL	1.01	NL
Digital Audio 0x0A	NL	NL	NL	1.01	1.01	1.02	1.03	1.03	1.02
Storage 0x0C	NL	NL	NL	NL	1.01	NL	1.02	1.02	1.02
Location 0x0E	NL	NL	NL	NL	NL	NL	NL	NL	1.00

Table 1-3 Most recent lingo versions for Apple devices, Table 2

Models	iPhone	iPhone 3G	4G nano	120 GB classic	2G touch	3G touch	iPhone 3GS	160 GB classic	5G nano
General 0x00	1.09	1.09	1.08	1.08	1.09	1.09	1.09	1.08	1.09

Models	iPhone	iPhone 3G	4G nano	120 GB classic	2G touch	3G touch	iPhone 3GS	160 GB classic	5G nano
Simple Remote 0x02	1.02	1.03	1.04	1.03	1.03	1.03	1.03	1.03	1.04
Display Remote 0x03	1.05	1.05	1.05	1.05	1.05	1.05	1.05	1.05	1.05
Extended Interface 0x04	1.12	1.12	1.14	1.13	1.12	1.14	1.14	1.13	1.14
USB Host Mode 0x06	NL	NL	NL	NL	NL	NL	NL	NL	NL
RF Tuner 0x07	NL	NL	1.01	1.00	NL	NL	NL	1.00	1.01
Accessory Equalizer 0x08	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Sports 0x09	NL	NL	1.01	NL	1.01	1.01	1.01	NL	1.01
Digital Audio 0x0A	1.02	1.02	1.03	1.03	1.02	1.02	1.02	1.03	1.03
Storage 0x0C	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02
iPod Out 0x0D	NL	1.00	NL	NL	1.00	1.00	1.00	NL	NL
Location 0x0E	1.00	1.00	NL	NL	1.00	1.00	1.00	NL	NL

Table 1-4 Most recent lingo versions for Apple devices, Table 3

Models	iPad, iPad 3G, iPad 2 (all models), iPad (3rd generation)	iPhone 4 (all models)	6G nano	4G touch
General 0x00	1.09	1.09	1.09	1.09
Simple Remote 0x02	1.02	1.03	1.07	1.03
Display Remote 0x03	1.05	1.05	1.05	1.05
Extended Interface 0x04	1.14	1.14	1.14	1.14
USB Host Mode 0x06	1.00	NL	1.00	NL
RF Tuner 0x07	NL	NL	1.01	NL
Accessory Equalizer 0x08	1.00	1.00	1.00	1.00
Sports 0x09	NL	1.01	1.01	1.01
Digital Audio 0x0A	1.02	1.02	1.03	1.02
Storage 0x0C	1.02	1.02	1.02	1.02

Models	iPad, iPad 3G, iPad 2 (all models), iPad (3rd generation)	iPhone 4 (all models)	6G nano	4G touch
iPod Out 0x0D	1.00	1.00	1.00	1.00
Location 0x0E	1.00	1.00	NL	1.00

The accessory should determine what features an attached Apple device supports by sending the General lingo command 0x4B, `GetiPodOptionsForLingo`. If the Apple device does not support that command, the accessory must extract the lingo version number from the Apple device. This protocol version information can be used to determine which features the connected Apple device supports. See "Command 0x0F: RequestLingoProtocolVersion" (page 131) for information about the `RequestLingoProtocolVersion` command. For past firmware versions, see Table C-1 (page 523). Table 1-5 (page 44) lists hardware and software features outside these protocols.

1.1 General Apple Device Features

Table 1-5 (page 44), Table 1-6 (page 45), and Table 1-7 (page 46) list Apple device hardware and software features that are not part of specific lingoes. The numbers in the tables show the firmware versions in which each of these features was introduced.

Table 1-5 Features supported by specific Apple device firmware and OS versions, Table 1

Features	Software versions										
	3G	mini	4G	4G (color)	nano	5G	2G nano	classic	3G nano	iPod touch	iPhone
Serial autobaud on framing errors	—	1.4.0	3.1	1.1	1.0	1.0	1.0	1.0	1.0	1.1	1.1
Display notification on unsupported iAP accessory attach	—	—	—	—	1.0	1.0	1.0	1.0	1.0	1.1	1.1
Apple device detects detach for all valid R_{ID} values	—	1.4.0	3.1	1.2	1.0	1.0	1.0	1.0	1.0	1.1	1.1
Authentication Version 1.00	—	—	—	—	1.0	1.0	1.0	1.0	1.0	1.1	1.1
Authentication Version 2.00	—	—	—	—	1.2	1.2	1.0	1.0	1.0	1.1	1.1
Video browsing[2]	—	—	—	—	—	1.2	—	1.0.3	1.0.3	2.0	1.1
iTunes tagging	—	—	—	—	—	1.2.3[1]	—	1.0	1.0	2.1	2.1

Features	Software versions										
	3G	mini	4G	4G (color)	nano	5G	2G nano	classic	3G nano	iPod touch	iPhone
Nike + iPod cardio equipment	—	—	—	—	—	—	—	—	1.1.2	—	—
Communication with iOS applications	—	—	—	—	—	—	—	—	—	3.0	3.0

[1] Accessories must not use the Storage lingo over USB with the 5G iPod. Only UART transport is supported.

Table 1-6 Features supported by specific Apple device firmware and OS versions, Table 2

Features	Software versions							
	iPhone 3G	4G nano	120 GB classic	2G touch	iPhone 3GS	3G touch	160 GB classic	5G nano
Serial autobaud on framing errors	2.1	1.0.2	2.0	2.1.1	3.0	3.1	2.0.3	1.0.1
Display notification on unsupported iAP accessory attach	2.1	1.0.2	2.0	2.1.1	3.0	3.1	2.0.3	1.0.1
Apple device detects detach for all valid R_{ID} values	2.1	1.0.2	2.0	2.1.1	3.0	3.1	2.0.3	1.0.1
Authentication Version 1.00	2.1	1.0.2	2.0	2.1.1	3.0	3.1	2.0.3	1.0.1
Authentication Version 2.00	2.1	1.0.2	2.0	2.1.1	3.0	3.1	2.0.3	1.0.1
USB Device Mode audio output	2.1	1.0.2	2.0	2.1.1	3.0	3.1	2.0.3	1.0.1
Video browsing[2]	2.1	1.0.2	2.0	2.1.1	3.0	3.1	2.0.3	1.0.1
iTunes tagging	2.1	1.0.2	2.0	2.1.1	3.0	3.1	2.0.3	1.0.1
Nike + iPod cardio equipment	—	1.02	—	2.1.1	3.0	3.1	—	1.0.1
Communication with iOS applications	3.0	—	—	3.0	3.0	3.1	—	—
iPod Out	4.0	—	—	4.0	3.0	4.0	—	—
Wi-Fi network login sharing	—	—	—	—	5.0	5.0	—	—

Features	Software versions							
	iPhone 3G	4G nano	120 GB classic	2G touch	iPhone 3GS	3G touch	160 GB classic	5G nano
AssistiveTouch	—	—	—	—	5.0	5.0	—	—

Table 1-7 Features supported by specific Apple device firmware and OS versions, Table 3

Features	Software versions					
	iPad, iPad 3G	6G nano	4G touch	iPhone 4 (all models)	iPad 2 (all models)	iPad (3rd generation)
Serial autobaud on framing errors	3.2	1.0	4.1	4.0	4.3	5.1
Display notification on unsupported iAP accessory attach	3.2	1.0	4.1	4.0	4.3	5.1
Apple device detects detach for all valid R_{ID} values	3.2	1.0	4.1	4.0	4.3	5.1
Authentication Version 1.00	3.2	1.0	4.1	4.0	4.3	5.1
Authentication Version 2.00	3.2	1.0	4.1	4.0	4.3	5.1
USB Device Mode audio output	3.2	1.0	4.1	4.0	4.3	5.1
Video browsing[2]	4.2	—	4.1	4.0	4.2	5.1
iTunes tagging	3.2	1.0	4.1	4.0	4.3	5.1
Nike + iPod cardio equipment	—	1.0	4.1	4.0	—	5.1
Communication with iOS applications	3.2	—	4.1	4.0	4.3	5.1
iPod Out	4.2.1	1.0	4.1	4.0	4.3	5.1
Wi-Fi network login sharing	5.0	—	5.0	5.0	5.0	5.1
AssistiveTouch	5.0	—	5.0	5.0	5.0	5.1

[1] Because version 1.00 of Lingo 0x0A contained a bug that was corrected in version 1.0.1, accessories should attempt Digital Audio only with Apple devices whose Lingo 0x0A version is higher than or equal to 1.0.1. See Table 4-233 (page 346).

[2] With Apple devices that store and display video content (in addition to music), an accessory may choose to navigate the Apple device's hierarchy of stored videos. For details, see "Video Browsing" (page 67).

1.2 Audio and Video Output Preferences

Various Apple devices support different settings and routings for video output and digital audio/video synchronization. To achieve the best results, an accessory must query the attached Apple device for its capabilities and set its audio and video output preferences as specified in this section. For information about synchronizing digital audio and video streams, see "Audio/Video Synchronization" (page 347).

1.2.1 Video Output Settings

The iAP General lingo includes commands that let automotive head units and other accessories determine if the Apple device supports video output, and if so, to control several video output preferences. All Apple devices let an accessory control the video screen configuration and format. Newer Apple devices also let the user control the video output connection type, aspect ratio, closed caption enabling, and subtitle behavior, as shown in Table 1-8 (page 47).

Table 1-8 Video output capabilities

Capability	5G iPod	iPod classic; 3G, 4G, 5G, 6G nano	iPod touch, iPhone	2G, 3G, 4G touch, iPhone 3G, 3GS, iPhone 4 (all models), iPad (all models)
Widescreen signaling	No	Yes	No	
Composite (interlaced) output	Yes	Yes	Yes	
S-Video (interlaced) output	Yes	Yes	Yes	
Component (interlaced) output	No	No	Yes	No
Component (progressive) output	No	Yes	No	Yes
4:3 fullscreen aspect ratio	Yes	Yes	Yes	
16:9 widescreen aspect ratio	No	Yes	Yes	
Closed captioning	No	Yes	Yes	
Subtitle display	No	Yes	No	
Display resolution	640 x 480	NTSC: 720 x 480; PAL: 720 x 576		

An accessory can determine if the Apple device supports video output by sending a `GetiPodOptionsForLingo` command. If the Apple device does not support that command, the accessory may send a `GetiPodOptions` command. If the Apple device returns an `iPodAck` command with a Bad Parameter status, or sends a `RetiPodOptions` command with the Video Option bit (bit 0) clear, the Apple device does not support video. If the video option bit is set, the Apple device at least supports video output, screen configuration, and video format preferences. The accessory can determine if the Apple device supports other preferences by sending a `GetiPodPreferences` command for each preference. If the Apple device returns an `iPodAck` command with Bad Parameter status, that preference is not supported (see "Command 0x29: GetiPodPreferences" (page 151)).

Note: Accessories must use iAP commands to request analog audio output from an Apple device.

1.3 Power Management

Accessories that provide power to Apple devices must specify how much power they make available, as documented in *MFi Accessory Hardware Specification*. An accessory may choose to use the General lingo command `SetAvailableCurrent`; if it does so, the command must be sent within 1 second after authentication completes and every time there is a change in the amount of power the accessory provides. Starting with iOS 5.0, an accessory may also tell the iOS device whether to draw power from the device's internal battery or from the accessory by sending a General lingo `SetInternalBatteryChargingState` command. This lets an accessory with its own battery optimize an attached iOS device's battery usage.

1.4 Status Notifications

Both Apple devices and their attached accessories can send asynchronous status notifications to each other, using General Lingo commands.

1.4.1 Status Notifications From Apple Devices

Most Apple devices are capable of generating asynchronous status notifications to inform accessories of changes in device state or status. Notifications of one type, Flow Control, are enabled by default if the accessory uses the Identify Device Preferences and Settings (IDPS) process, as specified in "Identification" (page 93). Accessories that use `IdentifyDeviceLingoes` instead of IDPS must enable Flow Control notifications explicitly. Receiving Flow Control notifications lets accessories cope with situations where the Apple device's incoming queue is full and it is unable to accept more packets.

To work reliably with all past and future Apple devices, an accessory should first send a `GetSupportedEventNotification` command to determine which notification types the Apple device supports. The Apple device can respond in one of two ways:

- The Apple device can return an `iPodAck` command with a Bad Parameter error. This means that the Apple device does not support the `GetSupportedEventNotification` command. The accessory may then send a `SetEventNotification` command, passing the Flow Control bit and any other bits in the returned bit mask. The Apple device may respond in one of two ways:
  - If the Apple device responds with a successful `iPodAck` command, it means that it supports Flow Control notifications and the accessory should be prepared to handle them.
  - If the Apple device responds with an `iPodAck` command that passes a Bad Parameter error, it means that it does not support notifications.

- The Apple device can return a `RetSupportedEventNotification` command with a notification bitmask indicating which notifications it supports. The accessory may then send a `SetEventNotification` command, passing the FlowControl bit plus any other bits chosen from the returned bitmask.

Accessories must rely on notifications to infer the state of an Apple device; repeated polling or querying of device state is not allowed. This approach minimizes traffic across the iAP transport link while reducing processor and power usage on the Apple device, leaving more resources available for activities such as media playback and application execution.

Unless explicitly stated elsewhere in this specification, sending any iAP command whose name begins with `Get` counts as an attempt to poll or query the Apple device's state, for which the following accessory actions are not allowed:

- Sending a `Get` command without previously registering with the Apple device for notifications that provide the same information.
- Sending a `Get` command within 1 second after receiving a notification from the Apple device that provided the same information.
- Sending a `Get` command within 1 second after receiving the corresponding `Ret` command from the Apple device.

These rules apply in all situations, including user actions such pressing buttons on an accessory. The following actions are particularly important to regulate in accessory design:

- Using the Display Remote lingo `GetiPodStateInfo` command to query the Apple device's `TrackTimePosMs`, `PlayStatus`, `Shuffle`, `Repeat`, or `TrackTimePosSec` state.
- Sending a Display Remote lingo `GetPlayStatus` command.
- Sending an Extended Interface `GetShuffle`, `GetRepeat`, `GetPlayStatus`, or `GetIndexedPlayingTrackInfo` command.

1.4.2 Status Notifications From Accessories

An accessory can use iAP accessory status notifications to inform an Apple device of changes in the accessory's state, such as error or fault conditions or the availability of Bluetooth-capable components (see "Bluetooth Autopairing and Connection Status Notifications" (page 55)). To use status notifications, the accessory must tell the Apple device during the authentication process that it is able to generate them. After accessory status notifications are enabled, the Apple device sends the accessory a `SetAccessoryStatusNotification` command with a list of the notifications that it wants to receive. From then on, the accessory must send an `AccessoryStatusNotification` command to the Apple device every time there is a change in its state corresponding to one of the listed notifications.

1.5 Accessory Control of iOS Devices

Accessory developers must take into account these current limitations on the ability of accessories to control an iOS device.

- An accessory cannot navigate the home screen of an iOS device.
- An accessory cannot unlock an iOS device's screen once it is locked.
- When the iOS device's screen is locked, most iAP commands continue to operate, but some Simple Remote lingo commands (such as the menu, select, and arrow button controls) produce no visible effect.
- Accessories are responsible for adjusting video out settings if they are not entering or using Extended Interface mode. If it needs to display video outside the display on an iOS device, an accessory must set the video out preference to On. Extended Interface mode automatically makes this setting.

1.6 Accessory Launching of iOS Applications

A compatible accessory can request that an iOS device launch an application on its behalf. This action can improve the user's experience by automatically starting up the accessory's companion app when the accessory establishes communication with the iOS device.

For an accessory to use this autolaunch feature on Apple devices running iOS 5.1.1 or earlier, all of the following conditions must be met:

- The iOS device's display must be On.
- The iOS device must be unlocked.
- Either the iOS device's Home screen or its iOS Music app must be frontmost.

An accessory sends the General lingo command `RequestApplicationLaunch` (0x64) to launch an application on an iOS device. The accessory passes an Application ID string, such as `"com.mycompany.myapp"`, to specify which application to launch. The iOS device replies with an `iPodAck` command that returns the application launch status. A status of OK (0x00) only indicates that the launch is possible; it does not guarantee that the application is running at that time. A status of Command Failed indicates that the application either does not exist on the iOS device or that the iOS device is in a condition that prevents the launch. The Accessory must not retry the `RequestApplicationLaunch` command if a Command Failed status is returned.

After its `RequestApplicationLaunch` command is successfully acknowledged, the accessory must not assume that the requested app is actually running. Instead, the accessory must take one of these actions:

- Wait to receive a General lingo `OpenDataSession` command for an External Accessory protocol that it is expecting to use; or
- Register for and receive both `NowPlayingApplicationBundleName` and `NowPlayingApplicationDisplayName` notifications. For details, see "Interaction with iOS Media Applications" (page 54).

1.7 Accessory Communication With iOS Applications

This section describes the communication process between accessories and iOS, and specifies the iAP commands that an accessory can use to open and maintain communication with an application.

Note: Before it can communicate with any application, an accessory must be identified through the IDPS process specified in "Identification" (page 93) and must authenticate itself using Authentication 2.0B, as described in "Authentication" (page 103). Using IDPS also requires enabling transaction IDs in iAP commands, as described in "Transaction IDs" (page 110).

If the iOS device does not support IDPS, the accessory must send it an `IdentifyDeviceLingoes` command with all fields set to 0xFF. This causes the iOS device to display an "Accessory not supported" message.

The IDPS process lets an iOS device match its applications with its attached accessories, making it possible to open communication between them. Once an accessory has been identified and authenticated, the process described below can create a route for two-way data transmission between it and compatible applications.

The communication process between an accessory attached to an iOS device and an application running on the device requires both accessory iAP commands and application programming. The iAP side of the process is described in this section. The iOS side uses the External Accessory Framework, which is described in Apple's Software Development Kit for iOS.

1.7.1 Setting Up a Communication Session

To communicate with an application, an accessory must first expose certain protocol identifiers by sending `protocolString` values to the iOS device; see Table 3-76 (page 167). This information is used to establish communication channels to the application.

The accessory must also send the Apple device an Apple-assigned `BundleSeedIDString` value, as shown in Table 3-77 (page 167). This string identifies the publisher of the preferred application designed to be used with the accessory. It is a unique ten-character string that is part of the App ID which Apple assigns to every application through the iOS Provisioning Portal. For the best user experience, application publishers should associate that `BundleSeedIDString` value with only one app on the App Store. The accessory sends this value during its startup identification, as described in "Identification" (page 93).

The design and maintenance of communication protocols between accessories and applications are entirely the responsibility of the developers of these products. Apple makes no attempt to secure protocol name space or provide for communication security between accessories and applications. Protocol names must be in reverse-DNS format and must be associated with domains that are registered with the Internet Corporation for Assigned Names and Numbers (ICANN), so that Apple can rely on each protocol having a unique owner. Apple does not require a developer to be the owner of the domain name used, but does require that the developer have permission to use the domain name for the protocol. For suggestions on protocol design, see "Communication Protocol Design Hints" (page 52).

When an application requests access to an accessory, using the iOS External Accessory Framework, the iOS device sends it an `OpenDataSessionForProtocol` command. As a part of this notification, `sessionID` and `protocolIndex` values are sent to the accessory, so it can identify incoming iAP commands for this session. A session ID is included in all iAP data transfer commands, so that multiple sessions may run concurrently. The accessory should accept the connection with an `AccessoryAck` response.

When the application has finished with the accessory, or the iOS device has determined that communication is no longer needed, a `CloseDataSession` command is sent to the accessory. The accessory must cease communication through the `sessionID` that was just closed. At this time the accessory can implement any power-saving features it may have.

1.7.2 Data Flow to and from the iOS Device

The iAP command `iPodDataTransfer` is used to communicate information to the accessory from the application and `AccessoryDataTransfer` to communicate information to the application from the accessory. Both iAP commands contain a `sessionID` parameter to identify which channel they are traveling through. The message delivery order is first in, first out. End-to-end delivery of data is normally reliable, but not absolutely guaranteed, so developer protocols must be designed to recover from loss of data. For information about iAP transports, see "Protocol Transports" in *MFi Accessory Hardware Specification.* For the rules that must be followed during a communication session, see "Command 0x42: AccessoryDataTransfer" (page 177) and "Command 0x43: iPodDataTransfer" (page 178).

1.7.3 Backgrounded Applications

Beginning with iOS 5.0, iOS applications that meet certain criteria may maintain communication with accessories while the applications are backgrounded. Application developers must consult Apple's iOS Developer Documentation for details on writing the required code. Accessory designs do not need to be modified to take advantage of this feature. But since not all Apple devices support backgrounded accessory connections, accessories that claim to be compatible with Apple devices running older versions of iOS must not require backgrounded sessions to remain active.

1.7.4 Matching Accessories With Applications

The iOS device matches applications and accessories by using the preferences and protocol names communicated by the accessory during its IDPS session. If the accessory names multiple protocols, it will be deemed compatible with an application if the application supports at least one of them. In addition, the accessory's `BundleSeedIDString` value is used to identify preferred or default applications.

A `BundleSeedIDString` is required to properly pair the accessory to its preferred or default application on the iOS device. The application must be submitted to Apple as part of the accessory certification process. Apple assigns a `BundleSeedIDString` value to the application developer, who provides this information to the accessory developer for inclusion in the accessory's firmware.

1.7.5 Communication Protocol Design Hints

The owner or creator of a protocol for communication between accessories and applications must define the preferences that are required or optional. Each accessory and application must then negotiate their level of compatibility directly. Apple does not referee protocols, and protocol creators must ensure that their accessories and applications can verify that they support the same features. The following suggestions can help with protocol design:

- Ensure that the accessory can always resynchronize its data stream in the event of an error.
- The round-trip latency of commands may vary; do not draw inferences from propagation times. There is no guarantee of delivery timing over the communication link.
- After a connection has been established, clearly establish which end speaks first.
- Begin communication with an exchange of meta-information, such as confirmation of the protocol and version. Give both ends the opportunity to declare that they cannot proceed due to protocol incompatibilities.

- Ensure that the transaction order is clear and that both ends know their position in it. Packets within each stream are guaranteed to be ordered but not free of gaps between packets, so ensure that the stream can be resynchronized after a gap.
- Ensure that responses can always be matched to their corresponding requests. The simplest forms of reliable communication are call and response. Interleaved communication streams are more complicated and less desirable.
- Establish an unambiguous indicator of the start of each packet.
- Add security where needed, but recognize that it slows down and complicates each transaction.
- Ensure that the other end of each transaction has both received and understood the last packet.
- Ensure that each packet is received intact and error free by adding a CRC value, checksum, or the like.
- Ensure that incoming data can be cached until it can be processed.
- Try to make the protocol extensible for future needs.

1.7.6 Communication iAP Commands

The General lingo commands that support applications communicating with accessories are listed in Table 1-9 (page 53) and documented in detail in "General Lingo Commands" (page 124). All commands are protocol version 1.09 and require authentication 2.0B.

Table 1-9 Accessory communication commands

CmdID	Command name	Direction	Parameters:bytes
0x3F	`OpenDataSessionForProtocol`	Dev to Acc	`{transID:2, sessionID:2, protocolIndex:1}`
0x40	`CloseDataSession`	Dev to Acc	`{transID:2, sessionID:2}`
0x41	`AccessoryAck`	Acc to Dev	`{transID:2, ackStatus:1, cmdID:1}`
0x42	`AccessoryDataTransfer`	Acc to Dev	`{transID:2, sessionID:2, data:<var>}`
0x43	`iPodDataTransfer`	Dev to Acc	`{transID:2, sessionID:2, data:<var>}`
0x4A	`iPodNotification`	Dev to Acc	`{transID:2, notificationType:1, waitTime:4, overflowTransID:2}`

Note: The `transID` parameters in the foregoing table are described in "Transaction IDs" (page 110).

1.8 Interaction with iOS Media Applications

Some iOS applications that play video or audio content are able to receive remote control events from accessories and/or send metadata about currently playing content to accessories. These include both Apple-owned applications such as Music, Videos, or YouTube, and third-party media playback applications. See the article "Remote Control of Multimedia" in the iOS SDK for information on how to enable this feature in an iOS application.

To send remote control events to such applications, an accessory must implement only one of these two approaches:

- Use the Simple Remote lingo to send one of the following events in response to direct user input:

  Play/Pause
  Pause
  Play/Resume
  Next Track
  Previous Track
  Begin Fast Forward
  Begin Rewind

  Note: An accessory using this approach must limit itself to sending only these events.

- Signal the iOS device to enter Extended Interface Mode and use the `ContextButtonStatus` command to affect playback state. The accessory must also register for the `NowPlayingApplicationBundleName` notification. When that notification is received and its payload is the string `'com.apple.mobileipod'`, the accessory may use other Extended Interface Mode commands. If the payload is any string other than `'com.apple.mobileipod'`, the accessory must not use any other Extended Interface Mode commands that affect playback state.

To receive information about currently playing content from iOS Applications, the accessory must implement only one of these two approaches:

- If the accessory uses the Simple Remote lingo, it must use the Display Remote lingo and associated notifications.
- If the accessory signals the iOS device to enter Extended Interface Mode, it must use Extended Interface Mode notifications.

The accessory may also register for the `NowPlayingApplicationDisplayName` notification to display the human-readable name of the currently playing application.

The accessory must unconditionally accept the playback state reported by the Apple device, regardless of what playback control commands the accessory previously sent. Some iOS applications do not accept all playback control commands—for example, audio streaming applications that do not let the user return to a previous track.

WARNING: Some iOS apps can also receive remote control events from accessories via other means, such as the External Accessory Framework. This kind of playback control is strongly discouraged. When third-party apps run concurrently with the built-in Music app or other apps that conform to the protocol documented above, sending alternative remote control commands can cause accessory malfunctions.

If an accessory is currently interacting with a third-party app, the only way to resume playback from the built-in Music app is to send the Resume iPod play control command documented in Table 5-49 (page 429). This command makes the built-in Music app resume playback as the NowPlaying app. This situation is the only one where use of the Resume iPod command is permitted, and the Resume iPod command will work only on Apple devices running iOS 5.1 or later.

1.9 Bluetooth Autopairing and Connection Status Notifications

Starting with iOS 5.0, an accessory can automate the Bluetooth pairing process between an iOS device and any number of Bluetooth devices that work with the accessory. An example of such an accessory would be a speakerphone that can play both music streamed from the 30-pin dock connector and telephone audio over Bluetooth Hands-Free Profile (HFP).

The accessory initiates the Bluetooth autopairing pairing process when it is docked with the iOS device. It must first use the General lingo `RetAccessoryStatusNotification` command to inform the iOS device that it is capable of sending Bluetooth status notifications (see "Status Notifications From Accessories" (page 49)). Then when sending status notifications, using the `AccessoryStatusNotification` command, the accessory must provide detailed information about each Bluetooth device. See "Command 0x47: RetAccessoryStatusNotification" (page 180) and "Command 0x48: AccessoryStatusNotification" (page 180) for the parameters to be sent. If the iOS device has Bluetooth enabled, and it has not previously paired with the Bluetooth devices, it will initiate a pairing process for each one. The Bluetooth devices must be ready to complete these pairing processes.

Some iOS devices can report changes in the Bluetooth connection status for any of the Bluetooth devices belonging to an accessory. These notifications can be useful when the accessory wishes to determine whether it is connected to a particular iOS device via both Bluetooth and its dock connector. For example, an accessory that is capable of receiving both audio output from an iOS device via the Bluetooth A2DP/AVRCP profiles, and USB Host Mode Audio through a dock connector, can use notifications to determine if both routes come from the same iOS device. The accessory can then choose whether to start music playback over Bluetooth or the dock connector. Similarly, if an accessory determines that one iOS device is connected via the dock connector while a second iOS device is connected via Bluetooth A2DP/AVRCP, the accessory can treat the two connections as belonging to two different iOS devices.

Note: If it intends to receive music playback audio output from an iOS device through its dock connector, an accessory must not send Bluetooth AVRCP commands while using the dock connector to connect to the device. Receiving AVRCP commands makes an iOS device route all outgoing media playback audio over Bluetooth A2DP instead of through the dock connector.

Before using Bluetooth Connection Status notifications, an accessory must send `GetSupportedEventNotification` to determine if the connected iOS device supports them. Once support has been verified, the accessory must send `SetEventNotification` to the device to register for the desired notifications and then send `AccessoryStatusNotification` commands to inform the iOS device of its Bluetooth-capable components.

After receiving `AccessoryStatusNotification`, the iOS device sends `iPodNotification` commands indicating the connection status of each unique Bluetooth MAC address that it owns. It sends more `iPodNotification` commands to the accessory whenever any connection status changes.

The accessory must inform the iOS device whenever the status of any of its registered Bluetooth-capable components changes by sending it an `AccessoryStatusNotification` command.

1.10 Wi-Fi Network Login Sharing

Some Apple devices running iOS 5.0 or later can share Wi-Fi Network Login information with an accessory. The user must grant permission for the iOS device to share this information, and the iOS device provides it only for the Wi-Fi network that it is currently using.

Note: Accessories that use the Wi-Fi Network Login Sharing feature must meet the hardware requirements specified in *MFi Accessory Hardware Specification.*

To use this feature, the accessory firmware must implement the following interchange of iAP commands:

1. The accessory must send a `RequestWiFiConnectionInfo` command to the iOS device when the user presses a button on the accessory or otherwise takes direct action to initiate sharing of login information.
2. The iOS device will send an `iPodAck` response to the accessory. If the returned status is 0x00 (Success) the accessory must proceed to step 3. Otherwise, the accessory must discontinue its attempt to retrieve Wi-Fi connection information and try again only if and when the user takes direct action again as specified in Step 1.
3. The device will send a `WiFiConnectionInfo` command to the accessory. There may be a noticeable period of time between the `iPodAck` and `WiFiConnectionInfo` commands. After receiving the `WiFiConnectionInfo` command, the accessory must attempt to join the WiFi network using the information returned and tell the user whether the attempt was successful or not.

1.11 Extended Interface Mode

Extended Interface mode allows the user interface of an Apple device to be translated to other environments; for example, a vehicle entertainment system such as a dashboard radio, large-screen display, or steering wheel mounted remote control. The chapter "Extended Interface Mode" (page 397) provides further details and specifies the iAP commands that allow access to an Apple device's database and device management.

Among the basic features that Extended Interface mode provides for accessories are these two:

- It allows file-system and database-format independence for the interfacing accessory.
- It allows the interfacing accessory to remain ignorant of the encoders and decoders used to process digital audio information.

1.11.1 Entering Extended Interface Mode

An Apple device does not automatically enter into Extended Interface mode; the accessory must explicitly switch into and out of the Extended Interface mode. Receiving a valid `ReturnPlayStatus` command tells the accessory that the Apple device has entered Extended Interface mode and Lingo 0x04 commands can be used. The Apple device will return Lingo 0x04 `iPodAck` commands with a bad parameter status if the accessory is not communicating properly with the Apple device in Extended Interface mode.

Accessories may send Extended Interface command packets only after they have successfully identified and switched into Extended Interface mode. Accessories must still obey the other inter-packet and inter-command timing issues specified in "Command Timings" (page 212).

1.11.2 Using Extended Interface Mode

There are four General lingo (Lingo 0x00) commands that allow accessories to determine what mode an Apple device is in and to switch between the two major modes, Standard UI and Extended Interface. These commands were implemented to allow a accessory to switch between modes without having to unplug the accessory. Multilingo accessories must use these commands to switch into and out of Extended Interface mode.

Note: Multilingo accessories that support Extended Interface mode do not automatically switch into the Extended Interface mode after the identification process completes. These accessories must use the General lingo mode switching commands to explicitly switch into Extended Interface mode.

Figure 1-3 (page 71) lists the General lingo command codes for querying, entering, and exiting Extended Interface mode.

Table 1-10 General lingo commands for switching modes

Command ID	General lingo command	Protocol version	Requires authentication	
			UART transport	USB transport
0x03	`RequestExtendedInterfaceMode`	1.00	No	Yes
0x04	`ReturnExtendedInterfaceMode`	1.00	No	Yes
0x37	`SetUIMode`	1.09	No	Yes
0x06	`ExitExtendedInterfaceMode`	1.00	No	Yes

Two logical entities need to be managed while browsing and playing content in Extended Interface mode: the content Database Engine and the Playback Engine. The following sections describe those engines and give an example of command traffic between an accessory in Extended Interface mode and an Apple device.

Table 5-1 (page 398) lists the complete set of Extended Interface command IDs, noting which are relevant to the Database and Playback Engines, as well as which the protocol versions in which those commands were introduced. Unless otherwise noted, all subsequent protocol versions will support existing commands.

1.11.2.1 The Playback Engine

The Playback Engine is active when an Apple device is in a playback state, such as play, fast forward, and rewind. It has a special play list, called the Now Playing playlist, that is used to determine what track or content item will be played next. The `SelectDBRecord` command, with a track, audiobook, or GeniusMix category and a valid index, is used to transfer selected database items to the Now Playing playlist and start the player at a specified item within that list. Changes to the database selection before or after this command have no effect on the current playback.

1.11.2.1.1 Playback Behavior

An Apple device's behavior when it receives a `SelectDBRecord` packet may depend on the shuffle setting, the repeat setting, and on the kind of tracks in a playlist. Consider a playlist with the following tracks:

Track A
Track B
Track C
Track D
Track E

If shuffle mode is on, a `SelectDBRecord` packet with an index of 0 or greater causes the playlist to be shuffled randomly but with the first track set to the selected index. For example, with an index of 4, the playlist may be sent to the Playback Engine in the following order and play starts with Track E. If repeat is set to One, track E will repeat; if it is set to All, the entire playlist will repeat.

Track E
Track B
Track A
Track D
Track C

Note: The index numbers of tracks in the Playback Engine are always arranged in the current playlist order, starting with 0.

If shuffle mode is off, a `SelectDBRecord` packet with an index of 0 or greater causes the playlist to be sent to the Playback Engine in the current playlist order but with the first track set to the passed index. With an index of 2, using the current example, the playlist is sent to the Playback Engine in this order but play starts at Track C, proceeds to Track D, and so on:

Track A
Track B
Track C
Track D
Track E

Note: Shuffle mode affects the playing of podcasts only if the user has disabled their Skip When Shuffle attribute in iTunes before syncing the Apple device.

If Repeat is set to 0, playback will stop at the end of the playlist. If Repeat is set to 1, the track corresponding to the selected track index will repeat. If Repeat is set to All, the entire playlist will repeat.

In all the foregoing scenarios, the Apple device returns a successful `iPodAck(0)` packet to the accessory. The accessory should not assume that any tracks are playing, but should use `GetPlayStatus` to verify that playback has started.

Note: Commands in other lingoes may change the Apple device's playback status, including its player state, track position, and track index. Examples include the Simple Remote lingo command `ContextButtonStatus` and the Display Remote lingo command `SetCurrentPlayingTrack`. Accessories should request notifications (via `SetPlayStatusChangeNotification`) to coordinate itself with the Apple device's playback status.

1.11.2.1.2 Playback Status Notifications

If requested, an Apple device can generate playback status notifications and send them to the accessory whenever its playback status changes. See the `SetPlayStatusChangeNotification` and `PlayStatusChangeNotification` commands for details about setting up and receiving notifications.

IMPORTANT: Accessories that display information available from the `PlayStatusChangeNotification` command must always respond to the corresponding notifications. For example, an accessory that displays a list of tracks currently being played must respond to every "Playback engine contents changed" notification. Similarly, an accessory that displays track position must respond to the corresponding track time offset notifications. Continuous polling, using `GetPlayStatus`, is not an acceptable alternative.

Notifications are sent only when status changes occur, not at regular intervals. Accessories must accept the latest `PlayStatusChangeNotification` or `ReturnPlayStatus` command they receive as representing the current Apple device state, regardless of how long ago the command was sent. The accessory must never assume that the Apple device has stopped playing if no `PlayStatusChangeNotification` command has been received recently.

As a result of actions made by other accessories, an accessory may receive a `PlayStatusChangeNotification` command indicating an unrequested change in either the Track Index or Playback Engine Contents. In such cases, the accessory must query the Apple device again for information about the currently playing track and must not simply re-assert its previous track selection.

Note: Status notification settings persist across changes in Apple device power states.

1.11.2.2 The Database Engine

The Database Engine is always accessible when the unit is awake. It can be manipulated remotely and allows groups of content items to be selected, independently of the Playback Engine. This allows the user to listen to an existing track or playlist while checking the Apple device database for another selection. Once a different database selection is made, the user selection (the track or content playlist) is sent to the Playback Engine.

The commands such as `ResetDBSelection` and `GetNumberCategorizedDBRecords` are examples of commands that are used to manipulate the Database Engine. You can identify database commands from playback commands by the string “DB” or “Database” embedded in the command name.

1.11.2.2.1 Database Category Hierarchies

The database engine uses **categories** to classify music, videos and other records stored in the database. Possible categories are playlist, genre, media kind, artist, album, track, composer, audiobook, and podcast. A list of records can be assembled, based on the various selected categories, to create a user list of records (a playlist).

The database categories have a hierarchy by which records are sorted and retrieved. This category hierarchy has an impact on the order in which records must be selected. For example, if a low category, such as album, is selected first, followed by a higher relative category such as genre, the album selection is invalidated and is ignored. When creating a new set of database selections, the accessory must begin by resetting all database selections, using the `ResetDBSelection` or `ResetDBSelectionHierarchy` commands, and selecting the desired database categories from highest to lowest relative category. The category hierarchy, from highest to lowest (excluding podcasts), is shown in Table 1-11 (page 60).

Table 1-11 Database category hierarchy (excluding podcasts)

Category	Notes
All (highest level)	This is the state after a `ResetDBSelection` or `ResetDBSelectionHierarchy` command. No database categories are selected. If the `GetNumberCategorizedDBRecords` command is sent while in this state, it returns the total number of records for the requested category.
Playlist	When the `SelectDBRecord` command selects a playlist, all lower database category selections (genre or media kind, artist or composer, album, and track) are invalidated.
Genre or Media Kind	When the `SelectDBRecord` command selects a genre, all lower database category selections (artist or composer, album, and track) are invalidated.
Artist or Composer	When the `SelectDBRecord` command selects an artist or composer, all album and track category selections are invalidated.
Album	When the `SelectDBRecord` command selects an album, all track category selections are invalidated.
Track or Audiobook (lowest level)	When the `SelectDBRecord` command selects a track (music or video) or an audiobook, it is automatically transferred from the Database Engine to the Playback Engine. Selecting an audiobook in the Audiobook category does not affect the selection of a track in another category, and vice versa.

There are parallel hierarchies for podcasts and audiobook tracks. When the podcast category is selected, all track or audiobook selections are invalidated. If other categories (such as Artist) are then selected after the podcast category, the list of podcasts returned by `GetNumberCategorizedDBRecords` and `RetrieveCategorizedDBRecords` is affected. This behavior parallels that of playlists with multiple categories selected. The audiobook category always lists all audiobook tracks.

The default sort order for podcasts is alphabetical. For podcast episode tracks, the order is reverse chronological; that is, the most recent one is first.

Table 1-12 Database category hierarchy for podcasts

Category	Notes
All (highest level)	This is the state after a `ResetDBSelection` or `ResetDBSelectionHierarchy` command. No database categories are selected. If the `GetNumberCategorizedDBRecords` command is sent while in this state, it returns the total number of records for the requested category.
Podcast	When the `SelectDBRecord` command selects a podcast, all lower database category selections are invalidated.
Episode Track (lowest level)	When used in the context of podcasts, episode is synonymous with track. When the `SelectDBRecord` command selects an episode track, it is automatically transferred from the Database Engine to the Playback Engine.

1.11.2.2.2 Database Category Counts

The record count for a given database category is established either when the `GetNumberCategorizedDBRecords` command, or when the `SelectDBRecord` command is processed for that category. Depending on the previous database state and the order in which the categories are selected, the category record counts returned by the `GetNumberCategorizedDBRecords` command may be different.

After all database selections have been reset using the `ResetDBSelection` command, sending the `GetNumberCategorizedDBRecords` command for any category retrieves the total number of records matching that category in the entire Apple device database. In contrast, if one or more categories are already selected, the `GetNumberCategorizedDBRecords` command returns the number of records that match the both the specified category, as well as any already selected categories. As an example, assume an Apple device has the following four tracks in its database:

Genre	Artist	Album	Track
Electronica/Dance	Dirty Vegas	Essential Mix	Days Go By
R&B	Radiance	Pick 'n Choose	All Night
Hip-Hop/Rap	Coolio	It Takes A Thief	Fantastic Voyage
Electronica/Dance	New Order	Power, Corruption & Lies	Blue Monday

Imagine the following command sequence:

1. The database is reset using `ResetDBSelection`. All record counts are cleared.
2. The track record count returned by `GetNumberCategorizedDBRecords` is four, the total number in the database.
3. The genre Electronica/Dance is selected. The track record count returned by `GetNumberCategorizedDBRecords` is two. Two records match the Electronica/Dance genre category: the tracks by Dirty Vegas and New Order.

4. The artist Dirty Vegas is selected. Because the genre category is still selected, the track record count returned by `GetNumberCategorizedDBRecords` is one. One record matches both the Electronica/Dance genre category and the Dirty Vegas artist category: the track "Days Go By."

Note that in this example, each time a category is selected the track count is invalidated because it is in a lower category than the category selected. See "Database Category Hierarchies" (page 60) for details about the effects a category selection has on other category selections.

Retrieving the record count of a higher category does not cause the current selection to be invalidated. However, selecting one of the results of that count does cause an invalidation. For example, using the same Apple device database, a different command sequence produces different results:

1. The database is reset using `ResetDBSelection`.
2. The artist Dirty Vegas is selected. The Track record count returned by `GetNumberCategorizedDBRecords` for the Artist category is one, as there is only one track matching the Dirty Vegas artist category: the track "Days Go By."
3. The Genre record count is requested using `GetNumberCategorizedDBRecords`; this also returns a record count of one. Had the database included multiple Dirty Vegas songs from the same genre, then the returned Genre record count would still have been one. Had the database included Dirty Vegas songs from multiple different genres, however, the returned Genre record count would have been greater than one.
4. The Electronica/Dance genre is selected, which results in the Dirty Vegas selection being invalidated. If the `GetNumberCategorizedDBRecords` command is used to obtain the record count for the Artist category, it returns two, the expected total number of artists matching the Electronica/Dance genre (Dirty Vegas and New Order).

1.11.2.2.3 The All Tracks Playlist

The Apple device's All Tracks playlist contains every track in the current hierarchy (either audio or video) that is stored on the Apple device and always resides at index 0x00 in the Playlist category.

Accessories can use the following steps to select and play all the tracks on the Apple device:

1. Call `ResetDBSelection()`
2. Call `GetNumberCategorizedDatabaseRecords(playlists)`. This returns the number of playlists available on the Apple device.
3. Call `SelectDBRecord(playlists, 0)`. This selects the All Tracks playlist.
4. Call `PlayCurrentSelection()`.

1.11.2.2.4 Nested Playlists

Version 1.13 and later of Extended Interface mode supports nested playlists. A playlist can contain either tracks or one or more other playlists, but not both. The number of playlist nesting levels is unlimited.

When navigating playlists using the Apple device's Playlist category, nested playlists are flattened and their contents are presented in the highest level playlist that contains them. For example, if playlist Adam contains two nested playlists, Bob and Carol, then the Playlist category displays three playlists: Adam, Bob, and Carol. The contents of playlists Bob and Carol are their tracks. The contents of playlist Adam is the concatenation of the tracks in playlists Bob and Carol.

The Apple device provides a Nested Playlist category to traverse playlists with their nesting hierarchy in place. In the foregoing example, only playlist Adam would be returned when traversing the Nested Playlist category after a `ResetDBSelection` command. Selecting playlist Adam would expose nested playlists Bob and Carol. Selecting Bob or Carol would expose no further nested playlists. A typical command sequence for traversing the hierarchy looks like this:

1. Call `ResetDBSelection`; all record counts are cleared.
2. Call `GetNumberCategorizedDBRecords`; a `NestedPlaylist` count of 1 is returned.
3. Call `SelectDBRecord`; select a `NestedPlaylist` index of 0 (playlist Adam).
4. Call `GetNumberCategorizedDBRecords`; a `NestedPlaylist` count of 2 is returned.
5. Call `SelectDBRecord`; select a `NestedPlaylist` index of 1 (playlist Carol).
6. Call `GetNumberCategorizedDBRecords`; a `NestedPlaylist` count of 0 is returned.

When the count of nested playlists returned by `GetNumberCategorizedDBRecords` is 0, the current playlist has no nested playlists. At this point the accessory should query the Apple device for the number and names of the tracks in that playlist.

Calling `SelectDBRecord` with a `NestedPlaylist` index of –1 traverses the hierarchy upward to the next parent playlist. If the current playlist has no parent playlist, passing index –1 has no effect.

1.11.2.2.5 Unexpected Database Changes

Wireless iTunes store access, rentals, Genius playlists, and other media features present opportunities for the Apple device's database contents to change without synchronizing to the iTunes database on a host computer. Category counts, category lists, track counts, tracks, and other media information may change at any time, including during sessions in Extended Interface mode. Playback engine counts and contents may also change as media are added to or removed from the Apple device.

Accessories must not expect the media information and content to remain the same throughout an Extended Interface session. Accessories must verify media information immediately before use to ensure that the data has not changed, especially when media information is cached locally on the accessory. Even then, the database or playback engine state may change without the Apple device informing the accessory. Accessories using Extended Interface mode must be programmed to recover from command errors and failures received unexpectedly from the Apple device, by retrying commands or by reidentifying themselves and entering Extended Interface mode again.

Note: An Accessory using Extended Interface mode must register for Database Changed notifications from the Apple device. Upon receiving each notification the accessory must take appropriate action, as specified above.

1.11.2.3 Transferring Album Art

The Extended Interface lingo includes several commands that support the transfer of album artwork from an Apple device to an accessory:

- `GetArtworkFormats`
- `RetArtworkFormats`
- `GetArtworkTimes`
- `RetArtworkTimes`
- `GetArtworkData`
- `RetArtworkData`

All album art image encoding is currently RGB-565, which can be transferred in both big- and little-endian formats. Artwork retrieval takes place in the following steps:

1. Retrieve the number of formats available for artwork on the Apple device using `GetArtworkFormats`. It is not necessary to call `GetArtworkFormats` more than once per session; these values will be static while the accessory is attached to the Apple device. However, there are no guarantees about the number of formats and which ones are available on a particular model or firmware version. Each `formatID` in `RetArtworkFormats` specifies both a pixel encoding, such as RGB-565 little-endian, and the image dimensions. All formats are fixed-size.

2. When the accessory wants to retrieve the artwork for a given track, it calls `GetIndexedPlayingTrackInfo` with an `infoType` of 7. This returns the count of artwork available for each `formatID` associated with the track. It is possible that a track may not have artwork for a particular `formatID` or that the number of images will vary by `formatID`. A given size of album artwork may be available for only one track in the database.

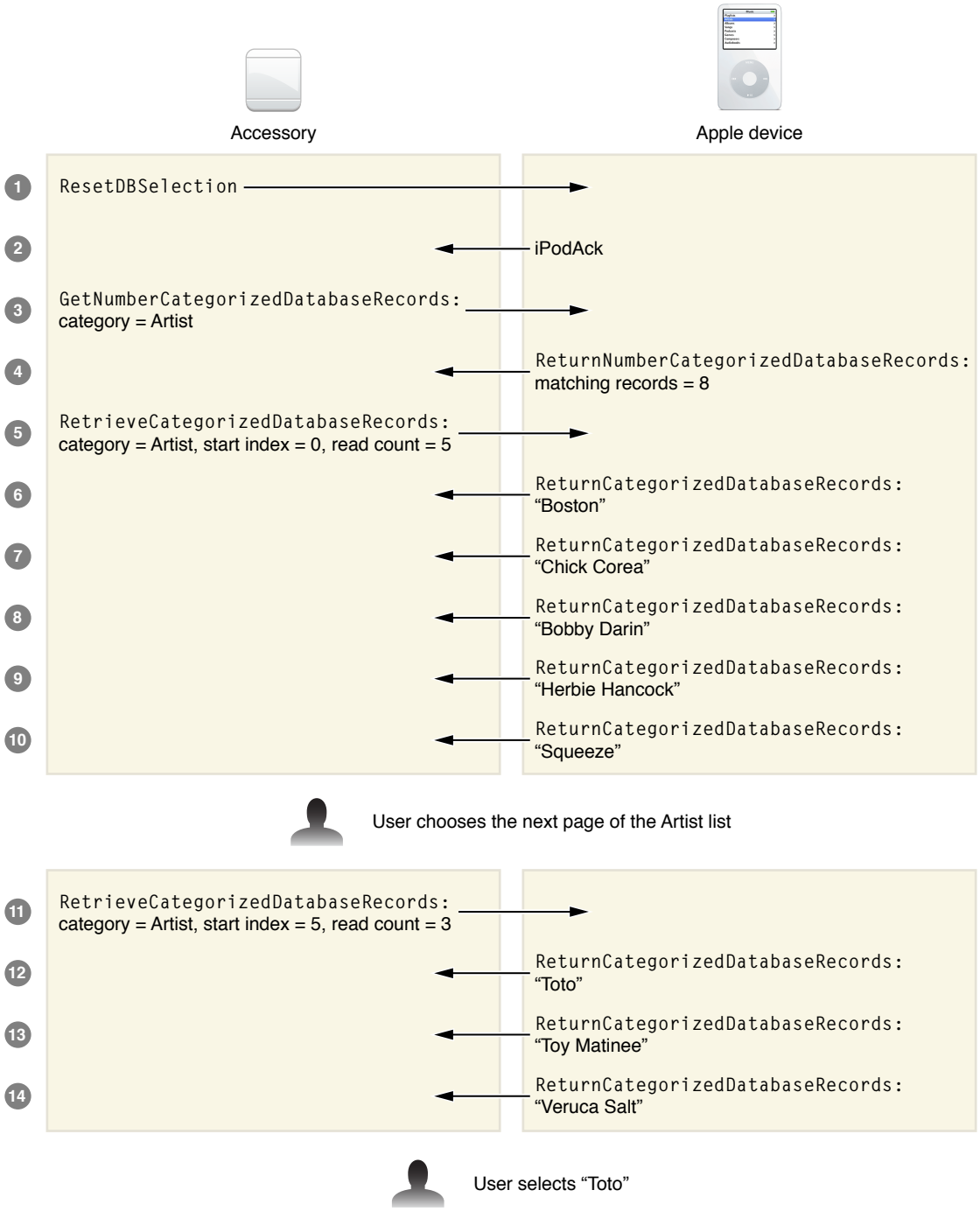
3. To retrieve the list of images associated with a given track and `formatID`, the accessory calls `GetArtworkTimes`. This command tells the Apple device to return the associated timestamp for each artwork. The timestamp indicates when the artwork should be displayed, expressed in milliseconds from the start of playback.

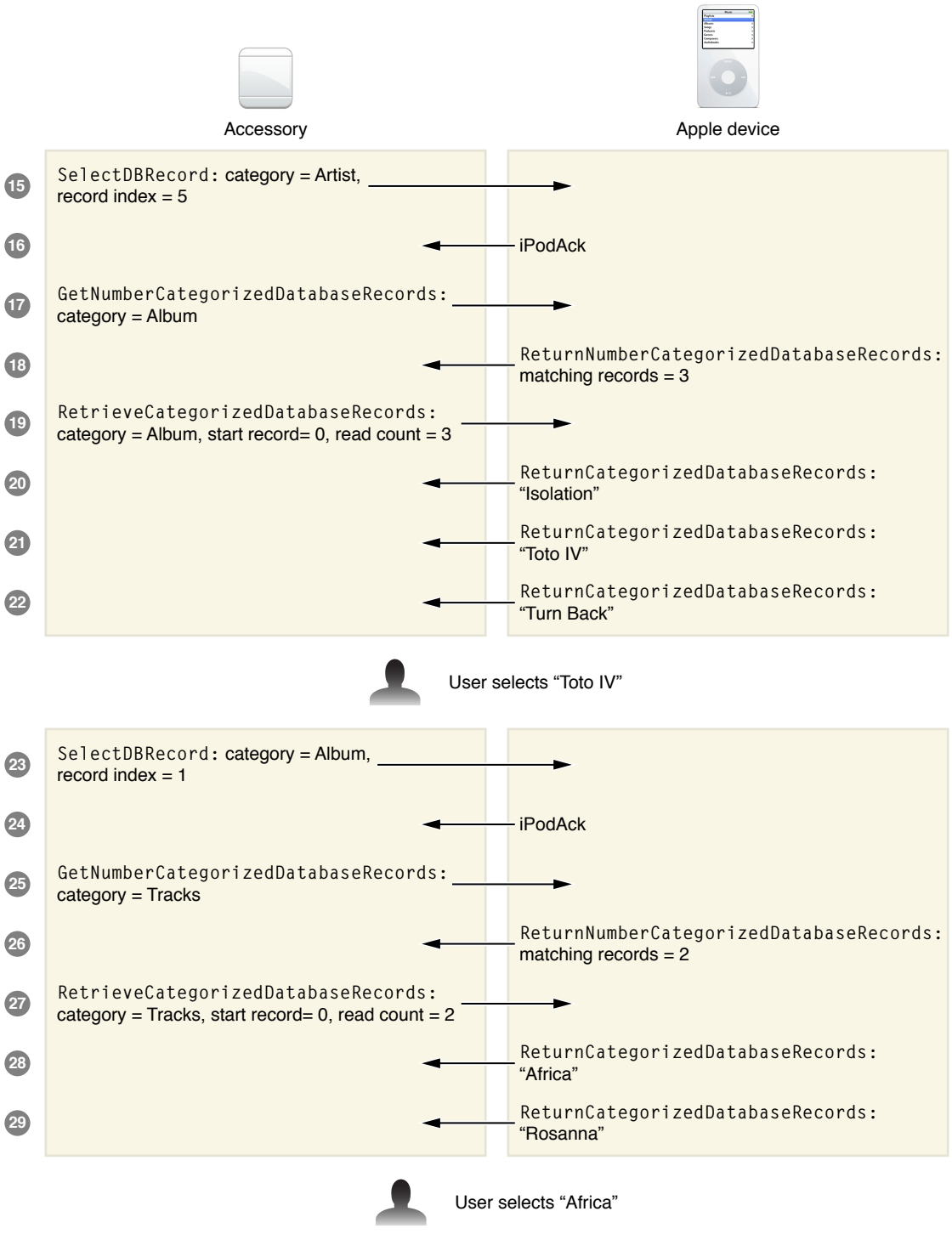
4. When the accessory wants to retrieve an individual piece of artwork, it sends `GetArtworkData` to the Apple device. This requires the accessory to specify a track, a formatID, and the timestamp of the desired image. The Apple device returns the specified artwork and the accessory can display it whenever it chooses.

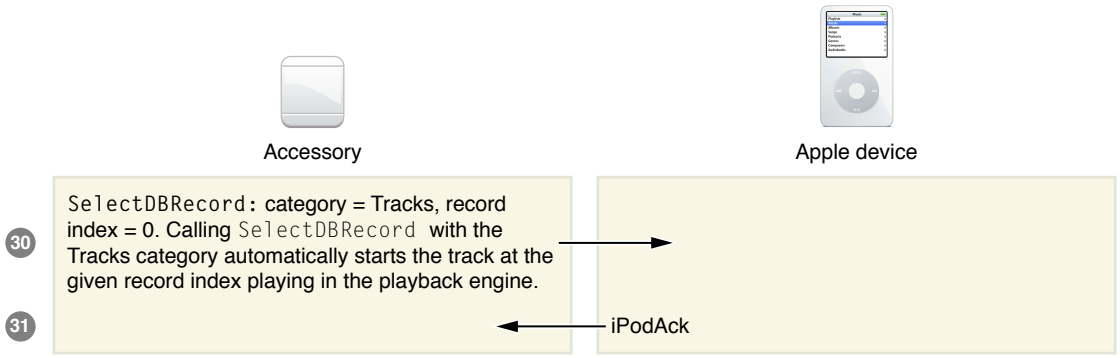
1.11.2.4 Using Extended Interface Mode: An Example

The following example of command traffic shows a conversation that might occur if the interfacing accessory has a paged menu of 5 choices; that is, a menu capable of displaying up to 5 options for the user to choose from.

Figure 1-1 An example interaction between an accessory and an Apple device in Extended Interface mode







1.11.2.5 Video Browsing

With Apple devices that store and display video content (in addition to music), an accessory may choose to navigate the Apple device's hierarchy of stored videos. To do this, the accessory must use the command `ResetDBSelectionHierarchy` to switch from navigating music tracks to navigating video tracks.

1.11.2.5.1 Video Playlists

A video playlist is a collection of tracks, created in iTunes, that contains one or more tracks playable as video. In contrast, an audio-only playlist does not contain any tracks playable as video. Video playlists may contain video-only tracks (such as movies) or tracks playable as either video or audio (such as video podcasts or music videos). Audio-only tracks are not visible and not playable when the Apple device is navigating the video hierarchy.

1.11.2.5.2 Video Navigation User Interface

Video track selection follows the Apple device user interface rules as closely as possible and behaves in the same way as audio track selection. In the video hierarchy, the menu item Genre is used to indicate media kind. The list of media kinds is dynamic and may be updated in the future to add, modify, or remove entries.

Once a media kind has been selected, the existing Artist, Album, and Song or Track categories may be used to further narrow the selection. For some media kinds, such as movies, a category may contain a single entry. In such cases, the accessory may choose to bypass this menu level. For example, with TV shows the Season menu on the Apple device is omitted if all the stored episodes are from the same season. Accessory designs that provide a video browsing user interface are encouraged to duplicate this behavior.

Video podcasts and music videos appear in both the music and video hierarchies. Other video tracks, such as movies and TV shows, appear only in the video hierarchy. Note that video playlists may contain audio-only tracks.

The Artist and Album categories do not apply to every media kind. For example, movies do not use these categories at the current time. When a category does not apply, the Apple device returns a count of 1 and displays a localized version of the word "All." The accessory may choose to bypass this menu level when displaying it to the user, as described above.

1.11.2.5.3 Navigating Video Playlists

The `ResetDBSelectionHierarchy` command with the video hierarchy type (0x02) selects the video hierarchy and invalidates all previous video database selections, such as `ResetDBSelection` in the audio-only hierarchy. As with the audio hierarchy, a video playlist record index of 0x00 designates the all video tracks playlist. If there are no video tracks on an Apple device, the all video tracks playlist will still be present, with a record count of 0.

After a `ResetDBSelectionHierarchy` command is sent to select the video hierarchy, all video playlists become visible. The video playlist count can be obtained using the `GetNumberCategorizedDBRecords` command with the playlist category (0x01). The video playlist records can be obtained using the `RetrieveCategorizedDatabaseRecords` command. A specific video playlist can be selected using the `SelectDBRecord` command.

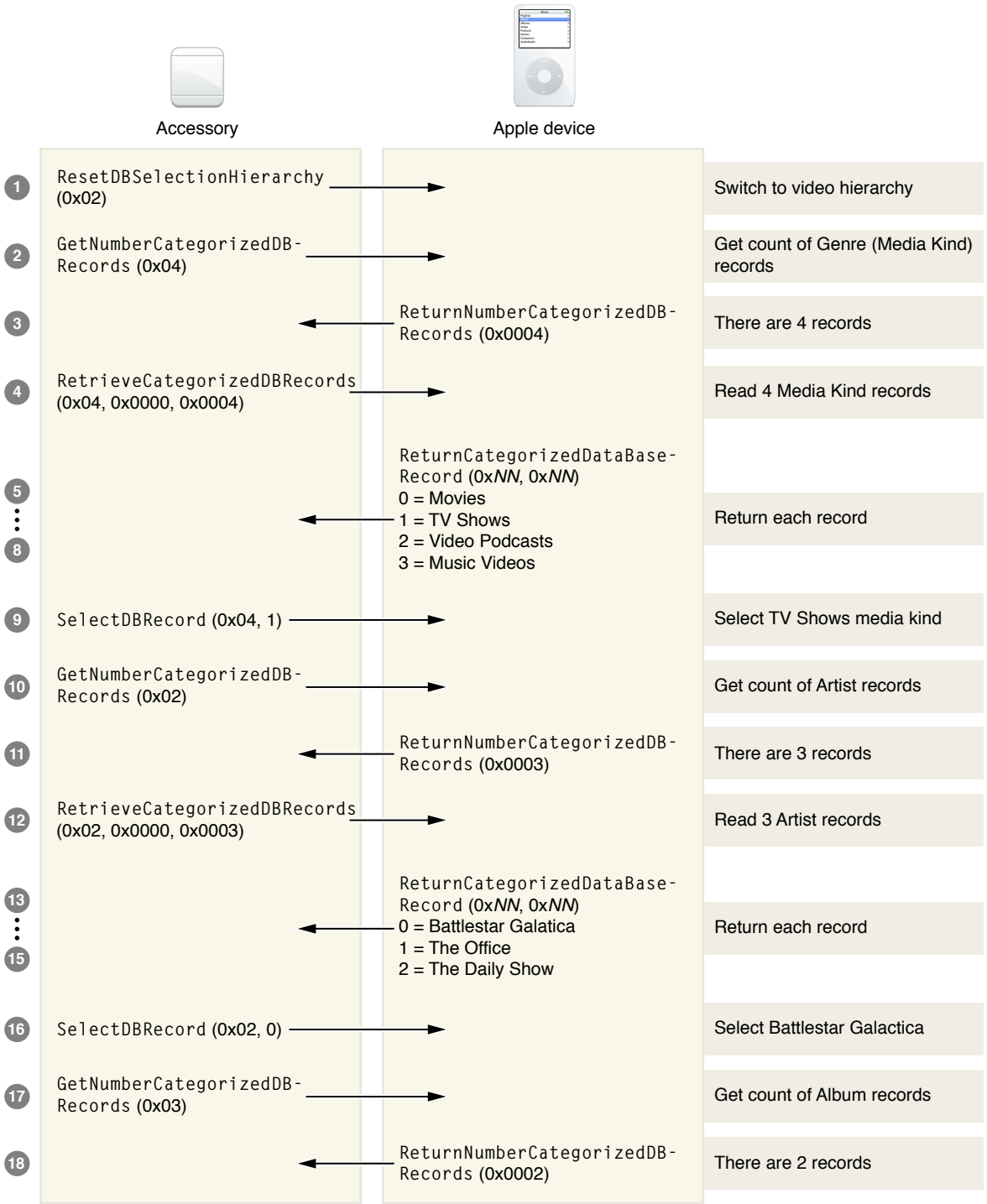
Compatibility Note: 5G Apple devices with firmware version 1.2.1 and earlier (before 11/2007) require that 1 be added to the `recIndex` values for music video artists. These Apple devices also do not filter out audio-only tracks while navigating the video hierarchy.

1.11.2.5.4 Video Selection Examples

Figure 1-2 (page 69) diagrams a typical interchange that might take place between an accessory and an Apple device as the user of the accessory navigates to a TV show stored on an Apple device.

TV shows and video podcasts use both the Artist and Album categories. Music videos return “All” as the only Album entry. Movies return “All” for both Artist and Album categories. However, this behavior may change in future versions of the Apple device firmware. Accessories must treat these categories as dynamic.

Figure 1-2 Navigating to a TV show: example of interactions between an accessory and an Apple device



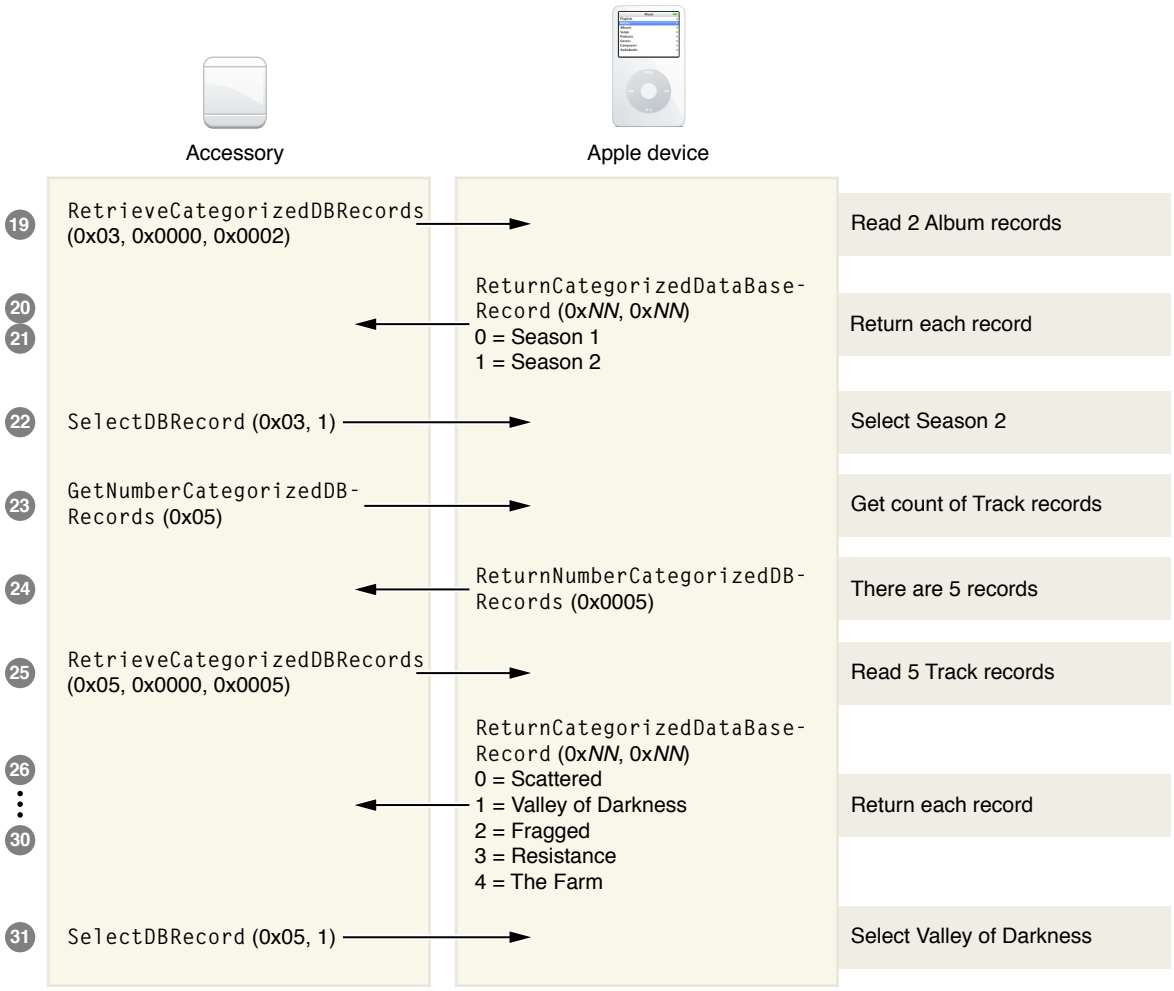
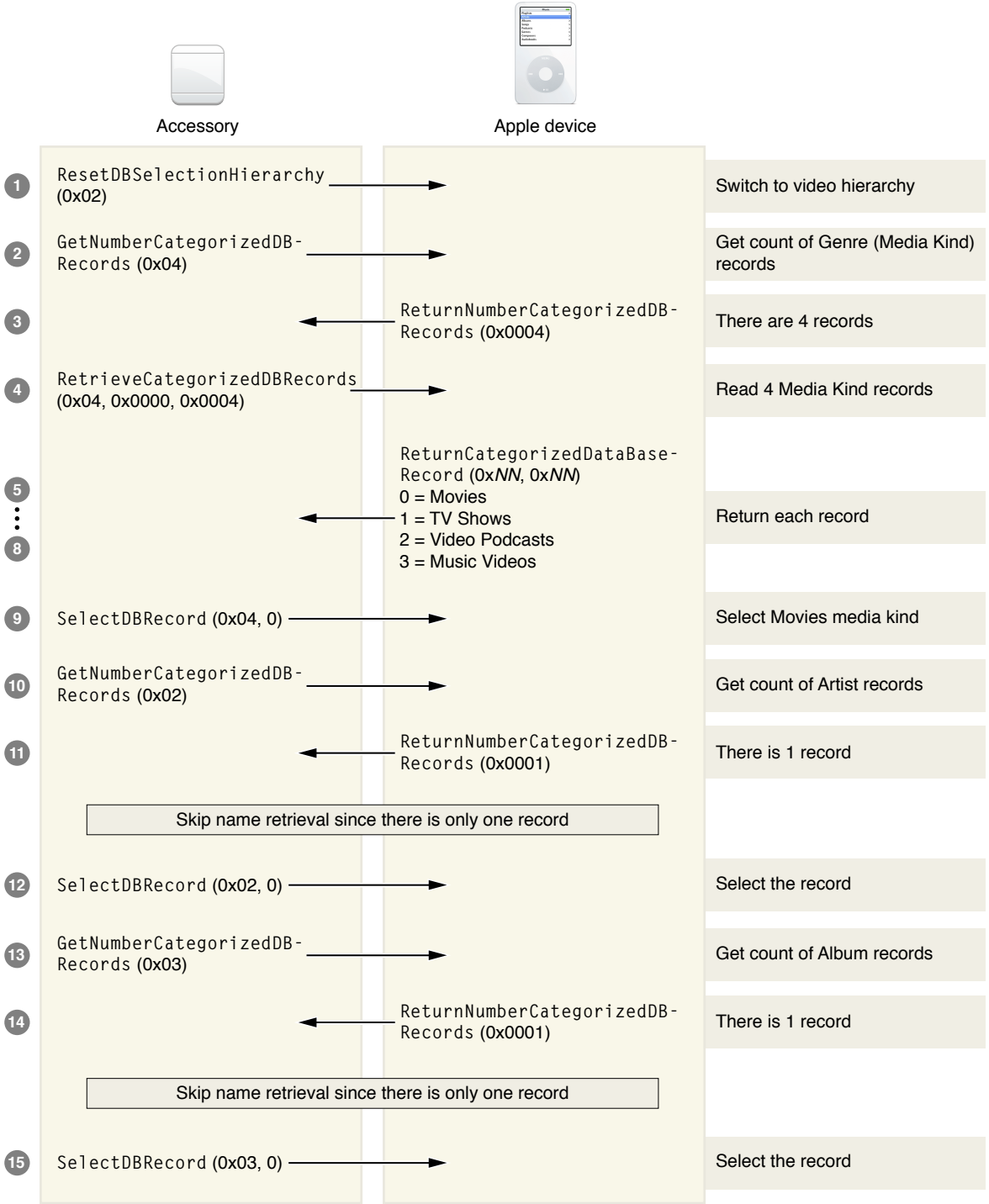
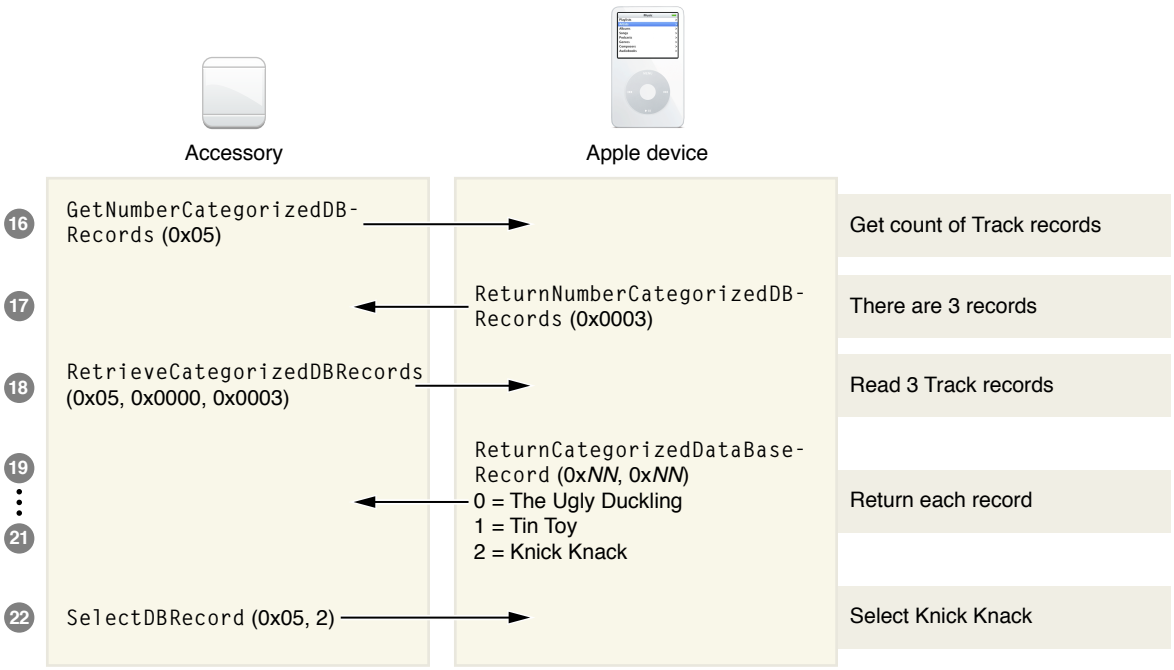


Figure 1-3 (page 71) diagrams the same kind of process with a movie. Note that in the case of the movie, the hierarchy search drops through two levels because those levels each contain only one record.

Figure 1-3 Navigating to a movie: example of interactions between an accessory and an Apple device





In video browsing, categorized DB records can be retrieved in any order, following essentially the same rules as navigating audio categories. It is not necessary to select the media kind. Table 1-13 (page 72) illustrates the process of selecting a playlist that contains video material.

Table 1-13 Selecting playlists containing video

Step	Action or command	Direction	Comments
1	Pass 0x02 (video) in `ResetDBSelection-Hierarchy`	Acc to Dev	Switch to video hierarchy.
2	Pass 0x01 (playlist) in `GetNumberCategorizedDBRecords`	Acc to Dev	Get count of playlists containing video. Any playlist that has at least one video track will be counted.
3	Receive 0x05 from `ReturnNumber-CategorizedDBRecords`	Dev to Acc	There are 5 records.
4	Pass (0x01,0x0000,0x0005) in `RetrieveCategorizedDatabaseRecords`	Acc to Dev	Get 5 video playlists starting with index 0.
5–9	Receive data from `ReturnCategorized-DataBaseRecord`	Dev to Acc	Return 5 record strings from the database.
10	Pass (0x01, 0x0001) in `SelectDBRecord`	Acc to Dev	Select video playlist index 1. Only video tracks will be displayed.

Following the steps shown above, the video track selection process can follow steps 16 to 22 shown in Navigating to a movie: example of interactions between an accessory and an Apple device (page 72).

1.11.2.6 Sample Extended Interface Session

Table 1-14 (page 73) shows a typical sequence of Extended Interface commands. This command sequence explores the Apple device's database and then returns information about a specific track.

Table 1-14 Typical extended interface commands

Step	Accessory command	Apple device command	Comment
The accessory identifies itself, is authenticated by Apple device, and enters extended interface mode.			
1	`GetDBiTunesInfo(0x00)`		Accessory requests database UID from Apple device. The UID is used by the accessory to determine if this Apple device database has been attached before.
2		`RetDBiTunesInfo`	The Apple device returns database UID. The accessory checks the UID to see if it has a match to previously cached DB contents.
3	`GetDBiTunesInfo(0x01)`		The accessory requests database last sync date/time.
4		`RetDBiTunesInfo`	The Apple device returns database last sync date/time. If DB has been cached by the accessory, the date/time is checked to see if the DB has been synced since the last time.
5	`GetDBiTunesInfo(0x02)`		The accessory requests database total audio track count.
6		`RetDBiTunesInfo`	The Apple device returns database total audio track count. If DB has been cached by the accessory, the audio track count is checked to see if the DB has changed since the contents were cached. A count of 0 (no tracks are present) can be used to gray out or remove audio options in the accessory's user interface.
7	`GetDBiTunesInfo(0x03)`		The accessory requests database total video track count.

Step	Accessory command	Apple device command	Comment
8		RetDBiTunesInfo	The Apple device returns database total video track count. If DB has been cached by the accessory, the video track count is checked to see if the DB has changed since the contents were cached. A count of 0 (no tracks are present) can be used to gray out or remove video options in the accessory's user interface.
9	GetDBiTunesInfo(0x04)		The accessory requests database total audiobook track count.
10		RetDBiTunesInfo	The Apple device returns database total audiobook track count. If DB has been cached by the accessory, the audiobook track count is checked to see if the DB has changed since the contents were cached.
11	ResetDBSelection		The accessory resets database selection before getting the total track count.
12		iPodAck	The Apple device acknowledges the ResetDBSelection command.
13	GetNumberCategorized-DBRecords(0x05)		The accessory gets the database track count.
14		RetNumberCategorized-DBRecords	The Apple device returns the database track count.
15	GetDBTrackInfo(0, trackCount, infoMask)		The accessory requests track information for all audio tracks. The infoMask tells the Apple device what types of information should be returned for each track.
16		RetDBTrackInfo(N, infoType, trackInfo)	The Apple device returns track index N information of type infoType.
...			
17		RetDBTrackInfo(N, infoType+t, trackInfo)	The Apple device returns track index N information of type infoType+t.
18		RetDBTrackInfo(N+1, infoType, trackInfo)	The Apple device returns track index N+1 information of type infoType.

Step	Accessory command	Apple device command	Comment
. . .			
19		RetDBTrackInfo(N+1, infoType+t, trackInfo)	The Apple device returns track index N+1 information of type infoType+t.
. . .			
20		RetDBTrackInfo(N+n, infoType, trackInfo)	The Apple device returns track index N+n information of type infoType.
. . .			
21		RetDBTrackInfo(N+n, infoType+t, trackInfo)	The Apple device returns track index N+n information of type infoType+t.
22	GetNumPlayingTracks		The accessory gets the playing track count.
23		RetNumPlayingTracks	The Apple device returns the playing track count.
24	GetPBTrackInfo(0, trackCount, infoMask)		The accessory requests track information for all audio tracks. The infoMask tells Apple device what types of information should be returned for each track.
25		RetPBTrackInfo(N, infoType, trackInfo)	The Apple device returns track index N information of type infoType.
. . .			
26		RetPBTrackInfo(N, infoType+t, trackInfo)	The Apple device returns track index N information of type infoType+t.
27		RetPBTrackInfo(N+1, infoType, trackInfo)	The Apple device returns track index N+1 information of type infoType.
. . .			
28		RetPBTrackInfo(N+1, infoType+t, trackInfo)	The Apple device returns track index N+1 information of type infoType+t.
. . .			
29		RetPBTrackInfo(N+n, infoType, trackInfo)	The Apple device returns track index N+n information of type infoType.
. . .			

Step	Accessory command	Apple device command	Comment
30		RetPBTrackInfo(N+n, infoType+t, trackInfo)	The Apple device returns track index N+n information of type infoType+t.
31	GetUIDTrackInfo(trackUID, infoMask)		The accessory requests track information for the specified track UID (the track UID was obtained via the GetDBTrackInfo or GetPBTrackInfo commands). The infoMask tells the Apple device what types of information should be returned for the track.
32		RetUIDTrackInfo(UID, infoType, trackInfo)	The Apple device returns track UID information of type infoType.
...			
33		RetUIDTrackInfo(UID, infoType+t, trackInfo)	The Apple device returns track UID information of type infoType+t.

1.11.2.7 Sample Extended Interface Session 2

Table 1-15 (page 76) shows a typical sequence of Extended Interface commands. This command sequence retrieves all information from the Apple device's media database and prepares a list of track UIDs for playback.

Table 1-15 Typical Extended Interface commands 2

Step	Accessory command	Apple device command	Comment
The accessory identifies itself, is authenticated by Apple device, and enters Extended Interface mode.			
1	ResetDBSelection		The accessory resets the Extended Interface browsing selection, in order to get information for all tracks.
2		iPodAck	The Apple device acknowledges the ResetDBSelection command.
3	GetDBiTunes-Info(DatabaseUID)		The accessory gets the database UID (can be used to detect if there have been any changes to the database).
4		RetDBiTunesInfo	The Apple device returns the database UID (e.g., 123456789ABCDEF).

Step	Accessory command	Apple device command	Comment
5	GetDBiTunes-Info(TotalAudioTrack-Count)		The accessory requests the total count of audio tracks.
6		RetDBiTunesInfo	The Apple device returns the count of audio tracks.
7	GetDBTrackInfo		The accessory requests information about all the tracks in the database: capabilities, title, artist, album, genre, composer, duration, and UID.
8		RetDBTrackInfo	The Apple device returns track info for track index 0.
9...		RetDBTrackInfo	The Apple device returns track info for all other tracks . . .
10	GetNumCat-DBRecords(Playlists)		The accessory requests the number of playlists.
11		RetNumCatDBRecords	The Apple device returns the number of playlists.
12	RetrieveCat-DBRecords(Playlist)		The accessory requests the names of all the playlists.
13		ReturnCatDBRecords	The Apple device returns the name of playlist index 0.
14...		ReturnCatDBRecords	The Apple device returns the names of all other playlists . . .
11	SelectDBRecord(Playlist, 1)		The accessory selects Playlist 1.
15		iPodAck	The Apple device acknowledges the SelectDBRecord command.
16	GetNumCat-DBRecords(Tracks)		The accessory requests the number of tracks in the playlist.
17		RetNumCatDBRecords	The Apple device returns the playlist track count.
18	GetDBTrackInfo		The accessory requests the UIDs of all tracks in the playlist.
19		RetDBTrackInfo	The Apple device returns the UID of track index 0.

Step	Accessory command	Apple device command	Comment
20...		RetDBTrackInfo	The Apple device returns the UIDs of all other tracks . . .
The Accessory uses the information gathered to initialize its own database and provide whatever UI is necessary to ask the user to select some tracks for playback.			
The user makes a selection or otherwise indicates that a specific list of tracks should start playback.			
21	PrepareUIDList(Section 0, maxSection NN)		The accessory sends the first section of a list of track UIDs.
22		iPodAck	The Apple device acknowledges the first PrepareUIDList command.
. . .			
23	PrepareUIDList(Section NN, maxSection NN)		The accessory sends the last section of a list of track UIDs.
24		iPodAck	The Apple device acknowledges the last PrepareUIDList command.
25	PlayPrepared-UIDList(firstUID)		The accessory tells the Apple device to start playing the list of tracks, starting with the track specified by firstUID.
The playback queue now contains the list of track UIDs, and the specified first track is playing.			

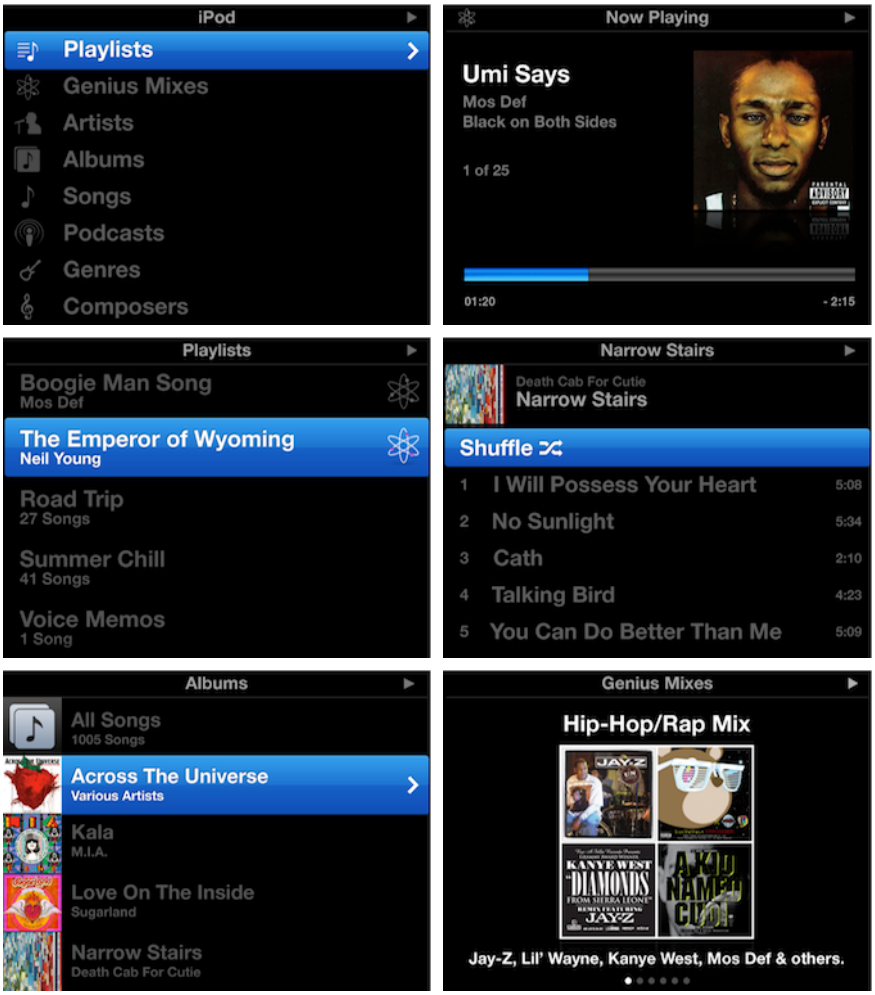
1.12 iPod Out Mode

iPod Out Mode is an Apple device operating mode that lets accessories interact with and control Apple devices through the native Apple user interface. While in this mode, the accessory supplies navigational inputs to the Apple device, using the Simple Remote or Display Remote lingoes, and displays video output from the Apple device on a separate display. Additional support is provided by the iAP accessory lingo specified in "Lingo 0x0D: iPod Out Lingo" (page 365).

The primary advantage of iPod Out over other operating modes (such as Extended Interface mode) is that accessories can focus on passing user navigational inputs to the Apple device and displaying the Apple device's output video signal. Accessories that use iPod Out automatically take advantage of new user interface features in future Apple software, because the Apple device exports the new features to the accessory's display.

Screen shots of sample iPod Out displays are shown in Figure 1-4 (page 79).

Figure 1-4 Sample iPod Out displays



Note: Accessory developers must observe the following requirements and cautions when using iPod Out Mode:

- Accessories must comply with the requirements of the section "iPod Out Accessory Requirements" in *MFi Accessory Hardware Specification*.
- For audio transport, the accessory may use either analog Line Out or digital audio over USB.
- The accessory must set its Video Out and analog audio transport preferences (if applicable) for iPod Out mode as part of the Identify Device Preferences and Settings (IDPS) process.
- Some Simple Remote and Display Remote commands have no effect on an Apple device while it is in iPod Out mode.
- Currently, NTSC at 4:3 aspect ratio is the only video output format supported in iPod Out mode.
- Currently, 640 by 480 is the only video output resolution in iPod Out mode.

The Apple device exits iPod Out mode whenever one of the following events occurs:

- It receives a `SetUIMode` command that passes a `UIMode` value other than 0x02.

- It receives an `EnterExtendedInterfaceMode` command. This method is **deprecated;** it may be used only if the Apple device returns an `iPodAck` command with a Bad Parameter result in response to `SetUIMode`.
- It is detached from the accessory.
- The IDPS authentication process begins again while the Apple device is in iPod Out mode and the accessory's authentication state is reset.

To use iPod Out mode, an accessory must identify itself using IDPS and must authenticate itself using Authentication 2.0.

1.12.1 Setting iPod Out Video Preferences

To use iPod Out mode, an accessory must be attached to an Apple device that supports video output. The iAP General lingo contains commands that let accessories query Apple devices for their video capabilities and set Apple device video preferences. All Apple devices that have video output support control of the video screen configuration and format. Newer Apple devices also support selection of the video output connection type, aspect ratio, closed captioning, and subtitles.

Before using iPod Out an accessory must send the Apple device an IDPS `iPodPreferenceToken` value for each of the following preference classes:

Video-out setting
Video signal format
Video-out connection

1.12.2 Supported Simple Remote and Display Remote Commands

While in iPod Out mode, Apple devices support a subset of the Simple Remote and Display Remote lingo commands. iPod Out accessories must not send any Simple Remote or Display Remote commands that do not appear in Table 1-16 (page 80).

Table 1-16 Simple Remote and Display Remote commands supported in iPod Out mode

Lingo	Command	Comments
Simple Remote (Lingo 0x02)	`iPodOutButtonStatus`	
	`RotationInputStatus`	
	`ContextButtonStatus`	Accessories may send only these button states (see Table 4-14 (page 226)): Play/Pause Next Track Previous Track Play/Resume Pause Begin Fast Forward Begin Rewind

Lingo	Command	Comments
Display Remote (Lingo 0x03)	SetRemoteEventNotification	
	GetRemoteEventStatus	
	GetiPodStateInfo	
	GetPlayStatus	
	GetIndexedPlayingTrackInfo	
	GetNumPlayingTracks	

1.13 VoiceOver

Accessories designed for users with special needs can work with the iOS VoiceOver feature by using iAP VoiceOver commands. All such accessories must check that those commands are supported by the Apple device and must identify for them during the IDPS process. Use of the VoiceOver iAP commands is not permitted if the accessory is not designed for users with special needs.

The Simple Remote lingo commands listed in Table 1-17 (page 81) can be used to navigate the iOS user interface and activate VoiceOver. Various parameters, such as VoiceOver's speaking rate and volume, can be queried and set using the `RetVoiceOverParameter` and `SetVoiceOverParameter` commands.

Note: Accessories must be prepared to deal appropriately with Simple Remote lingo `VoiceOverParameterChanged` commands from the iOS device at any time, because these commands may be sent asynchronously and multiple accessories may be active.

Table 1-17 Simple Remote lingo VoiceOver commands

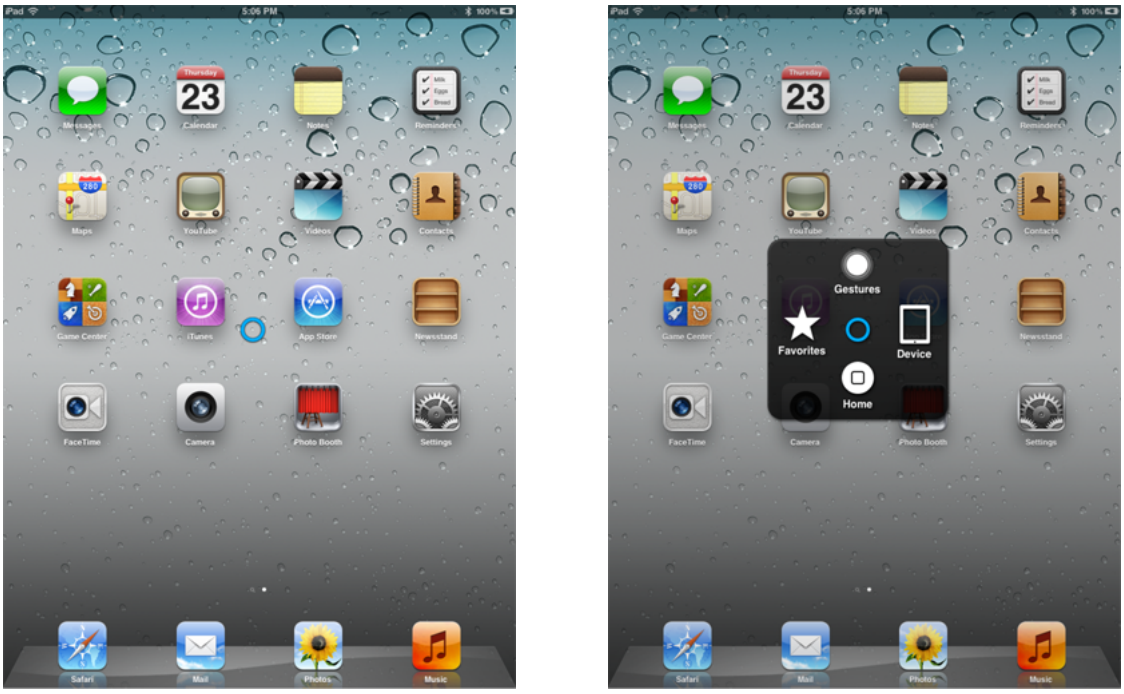
Cmd ID	Command name	Direction	Parameters:bytes
0x13	VoiceOverEvent	Acc to Dev	{eventType:1, eventData:<var>}
0x14	GetVoiceOverParameter	Acc to Dev	{paramType:1}
0x15	RetVoiceOverParameter	Dev to Acc	{paramType:1, paramValue:<var>}
0x16	SetVoiceOverParameter	Acc to Dev	{paramType:1, paramValue:<var>}
0x17	GetCurrentVoiceOverItemProperty	Acc to Dev	{propertyType:1}
0x18	RetCurrentVoiceOverItemProperty	Dev to Acc	{propertyType:1, propertyValue:<var>}
0x19	SetVoiceOverContext	Acc to Dev	{context:1}
0x1A	VoiceOverParameterChanged	Dev to Acc	{paramType:1, paramValue:<var>}

1.14 AssistiveTouch

Starting with iOS 5.0, accessories designed for users with special needs may augment the AssistiveTouch feature of iOS devices. AssistiveTouch is intended for users who have difficulty interacting with multitouch displays but can still touch a specific screen location either unaided or with the assistance of an accessory. Without an accessory, AssistiveTouch lets a user generate multi-touch gestures using one finger. Additionally, users with special needs who cannot touch the display at all can still interact with an iOS device using an accessory.

Accessories that use AssistiveTouch must meet certain hardware requirements, specified in *MFi Accessory Hardware Specification*. When the accessory is connected, an accessory-controlled cursor appears, letting the user simulate a single finger touch event (also sent by the accessory) at any point on the display. Multi-touch events and gestures can be generated by means of the context menu, as shown in Figure 1-5 (page 82). This menu can either remain on-screen at all times or be triggered by the accessory.

Figure 1-5 AssistiveTouch cursor and context menu



An Apple Device's iOS support for AssistiveTouch can be determined using `GetiPodOptionsForLingo`. The accessory must then enable the feature in the Apple device, using either an `iPodPreference` token or the `SetiPodPreferences` command.

The AssistiveTouch feature can be disabled and enabled while the accessory is still attached to an Apple device; this can be useful if the accessory needs to enter a different operating mode that does not provide an external pointing input to AssistiveTouch.

1.15 iTunes Tagging

Apple's iTunes Tagging feature lets a radio reception accessory provide a tagging function to the user. When the user tags a song being broadcast, the accessory captures the song's metadata and passes it along to an Apple device. For detailed accessory design requirements, see Appendix A, "iTunes Tagging Accessory Design" (page 467).

1.15.1 The iTunes Tagging Experience

To experience the capabilities of the iTunes tagging feature, the user of an Apple device typically goes through the following steps:

1. While listening to broadcast music on an accessory for an Apple device, the user hears a song and decides to tag it for future review or purchase by pressing a special button or other user interface element.
2. The accessory, equipped with a tag button or equivalent software action, stores metadata for the currently playing song. The user tags the song without needing to know anything about it or its originating source. The song information is stored in the accessory until an Apple device is connected to it.
3. When an Apple device is connected to it, the accessory transfers the tagged song information to the Apple device. The Apple device stores the data until the Apple device is synched with iTunes.
4. When the Apple device is connected to the user's computer, iTunes imports the tag data, analyzes it, and presents it to the user in the form of a tagged playlist. iTunes displays the song title, artist, album and other information for each tagged song.
5. Using iTunes, the user may save, review or delete any item in the tagged playlist and may easily purchase any of the listed songs through the iTunes store.

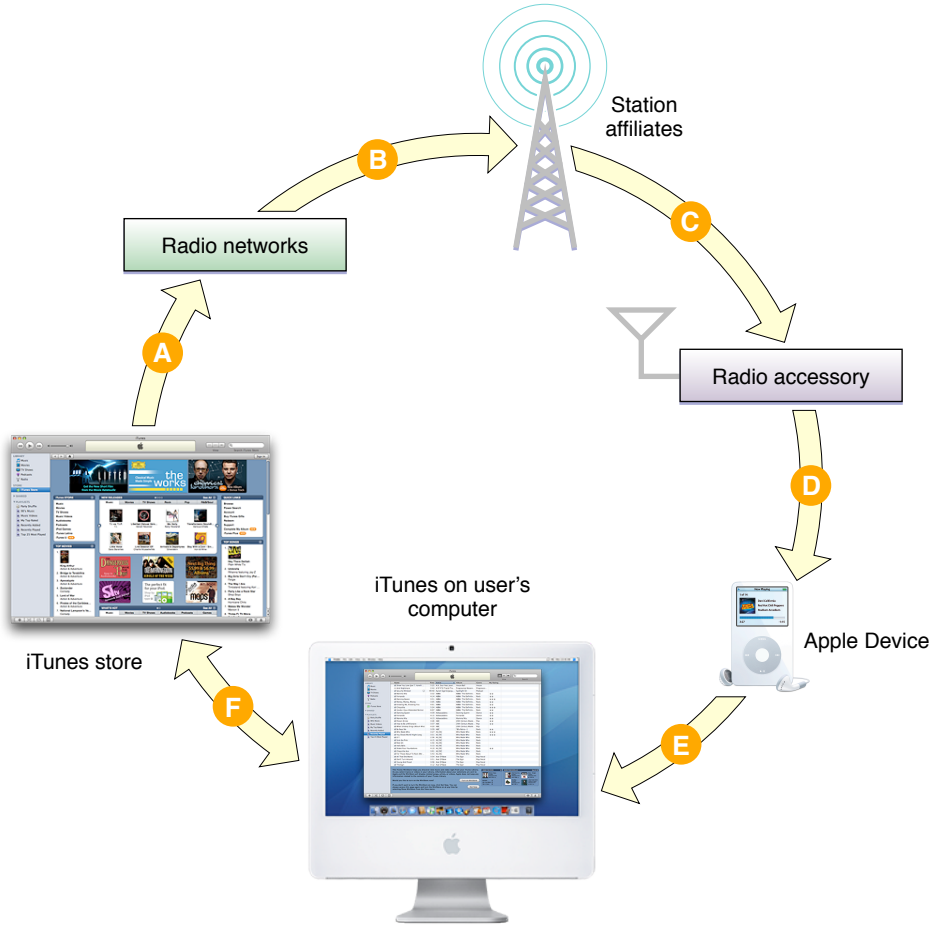
1.15.2 Tagging Feature Components

To implement the user experience described in the previous section, the iTunes tagging feature includes the following major components:

- The **iTunes store** creates and publishes unique identifiers for media available in its catalog, and allows users of accessories that support iTunes tagging to purchase tagged songs.
- **Broadcast networks and station affiliates** receive iTunes store data and broadcast it along with program music and other content.
- **Broadcast reception accessories for Apple devices** are designed to enable the tagging feature. These receivers may be portable, used in the home or in a car.
- An **Apple device** is used to transfer tags from broadcast reception accessories to iTunes.
- The **iTunes application** is a desktop media player for managing media content, including newly-discovered music via the iTunes tagging feature.

The data flows among the components of the iTunes tagging feature listed above are diagrammed in Figure 1-6 (page 84).

Figure 1-6 iTunes tagging feature data flows



Data flow	Description
A	An enterprise partner feed from Apple contains `iTunesSongID`s, track names, artists, and album names for the contents of the iTunes store.
B	Network data is sent to the station affiliates, including Apple-supplied data.
C	Affiliates broadcast metadata with each song, containing the song's `iTunesSongID`, track name, and artist name.
D	The broadcast reception accessory writes XML tag data to a file on an attached Apple device, using the iAP Storage lingo; the Apple device signs the file.
E	iTunes verifies and uploads the Apple device's tag files and presents a tagged music playlist to the user.
F	The user clicks a "Buy" button next to the tagged music and downloads the music from the iTunes Store.

1.16 Nike + iPod Cardio Equipment System

IMPORTANT: This technology is deprecated. Support for it is not guanteed in future Apple devices and it must not be used in new accessory designs.

Apple's Nike + iPod cardio equipment feature lets the user of an Apple device record workout data gathered by an attached sports or exercise accessory, while at the same time enjoying entertainment from the Apple device. An Apple device can support only one such accessory at any time, and that accessory must be attached using the 30-pin connector. Currently the 2G touch, the 3G nano, and the 4G nano support this system.

To use the capabilities of a Nike + iPod cardio equipment accessory, an Apple device user typically does the following:

1. The user connects a compatible Apple device to the cardio accessory, makes an entertainment selection, and selects either QuickStart or a specific workout type.
2. The cardio equipment downloads user preferences and other information from the Apple device.
3. If there is enough space on the Apple device to store the workout data, the cardio equipment displays the message "Recording workout to iPod." It opens a file and records data in real time.
4. The cardio equipment prompts the user to accept the weight setting retrieved from the Apple device or change it, in which case it stores the new weight on the Apple device.
5. When the workout ends, the cardio equipment closes the file on the Apple device and displays a closing message with a workout summary. It confirms that the workout has been recorded and tells the user how to view the results.
6. The user connects the Apple device to a computer and iTunes uploads the workout files to `nikeplus.com`, where the user can view the results.

For detailed accessory design requirements, see Appendix B, "Nike + iPod Cardio Accessory Design" (page 501).

1. Protocol Features and Availability

2. Protocol Core

This chapter covers the basics of the iPod Accessory Protocol (iAP), including accessory identification and authentication. It also describes the packet formats used for iAP commands.

The iAP is used to communicate in both directions, so every accessory must implement both sending and receiving capabilities. It should be possible to determine the direction (accessory to Apple device or Apple device to accessory) of a packet only from its contents. This means that no packet is valid for sending from both the Apple device and the accessory.

All accessories must be able to handle variable-length packets. An accessory must be able to accept a packet with extra data and ignore the unexpected bytes. At a minimum, the accessory must not lose sync to the packet signaling.

Most command packets generate a response, either an `iPodAck` command or a packet with data answering a query. Accessories should wait for the response before querying to see if the original request has been processed by the Apple device. In the Display Remote lingo, for example, when an accessory sends a `GetiPodStateInfo` command it should wait for the Apple device to return the corresponding `RetiPodStateInfo` (or `iPodAck`) command before sending a subsequent `GetPlayStatus` command.

Note: Unless otherwise stated in this specification, accessories must follow these timing rules:

- The accessory must wait at least 5000 ms for the Apple device to respond before retrying a command.
- When the Apple device sends a command for which it expects a response, the accessory must respond within 500 ms.
- If the accessory does not respond within 500 ms, it must not assume that the Apple device will retry the command.

2.1 Reserved Commands and Data

From time to time an Apple device may send an attached accessory a command that is reserved and not documented in this specification, or a documented command that has reserved data in its payload. Accessories must follow these rules when handling such reserved commands or data:

- When a data field is marked “Reserved” in this specification, accessories writing to it must set it to 0 (unless otherwise specified) and accessories reading it must ignore its value.
- If an accessory receives a command documented in this specification that passes reserved data, it must acknowledge successful receipt of the command, extract the data in it that is documented, and ignore the reserved data.
- If an accessory receives a command not documented in this specification, it should ignore it.

2.2 Transport Initialization

When attached, the accessory must detect the Apple device and initialize the transport by which it will communicate. This section describes the initialization processes for UART, USB Host mode, USB Device mode, and Bluetooth transports.

2.2.1 UART

When using UART transport, the identification process starts when the accessory detects current from the Apple device on the Accessory Power line of the 30-pin connector. The accessory must wait at least 80 ms and then transmit a sync byte. After sending the sync byte, the accessory must wait another 20 ms before starting the IDPS identification process.

Once accessory packet transmission has begun, the maximum time between transmitted data bytes is 25 milliseconds. If the inter-character delay exceeds 25 ms, the Apple device discards any packet characters already received, plus any remaining characters received received before the start of the next valid packet. The Apple device may exceed the 25 ms inter-character timing requirement in its outgoing packets when under heavy system load situations.

One known limitation exists when waking an Apple device from Sleep mode: the Apple device UART is not available for the first few milliseconds after waking from Sleep. If an accessory sends a packet to the Apple device while it is asleep, the first packet will be lost. Accessories must follow the steps below to wake an Apple device and ensure that the first packet is not lost:

1. Send a sync byte; this should wake the Apple device.
2. Wait for 20 ms.
3. Send the command packet with sync byte.

2.2.2 USB Host Mode

Certain Apple devices (see Table 1-6 (page 45)) can operate in USB Host mode, an operating mode in which the Apple device acts as a USB bus host and the accessory acts as a USB device. The following hardware details of USB Host Mode are documented in the section "Using an Apple Device as a USB Host" of *MFi Accessory Hardware Specification*, Version R6 or later:

- Design requirements for accessories that use USB Host Mode.
- How the accessory puts the Apple device into and out of USB Host Mode.
- Handling USB Device mode audio streams between the accessory and the Apple device.
- Handling USB MIDI.

USB Host mode may be advantageous to an accessory by letting it use low-cost USB chipsets to achieve faster data transfers with an Apple device.

2.2.2.1 Entering USB Host Mode on Startup

An accessory can be designed to use either hardware or firmware (but not both) to place an Apple device in USB Host Mode:

- Using hardware, the accessory can put the Apple device into USB Host mode as soon as it is connected. The required hardware configuration is described in *MFi Accessory Hardware Specification*. The accessory must immediately enter the Identify Device Preferences and Settings (IDPS) process and authenticate itself, using iAP commands over USB transport.
- If the Apple device supports USB Host Mode lingo, the accessory can complete the IDPS and authentication processes and then put the Apple device into USB Host mode by sending the iAP commands described in "Lingo 0x06: USB Host Mode Lingo" (page 283), using only UART transport. Table 2-1 (page 89) lists the iAP commands used to manage USB Host mode. After entering USB Host mode in this way, the accessory must continue to use UART transport for its exchanges of iAP commands with the Apple device.

Note: Because the USB Host Mode lingo replaces the deprecated USB Host Control lingo with the same lingo ID and version number (see "Command History of the USB Host Lingoes" (page 284)), the accessory must not use lingo ID information to test for USB Host Mode support.

Table 2-1 USB Host mode commands

Cmd ID	Command name	Direction	Parameters:bytes
0x00	`AccessoryAck`	Acc to Dev	`{cmdStatus:1, cmdIDOrig:1}`
0x04	`NotifyUSBMode`	Dev to Acc	`{usbMode:1}`
0x80	`iPodAck`	Dev to Acc	`{cmdStatus:1, cmdIDOrig:1}`
0x81	`GetiPodUSBMode`	Acc to Dev	none
0x82	`RetiPodUSBMode`	Dev to Acc	`{usbMode:1}`
0x83	`SetiPodUSBMode`	Acc to Dev	`{usbMode:1}`

Note: Once the Apple device is in USB Host mode it will remain in that mode regardless of whether the USB V_{BUS} supply is turned on or off. However, the Apple device will exit USB Host mode if it hibernates or the accessory reidentifies itself.

2.2.2.2 Upgrading from UART Transport to USB Host Mode

An accessory may start up sending iAP commands over UART transport and upgrade to USB Host Mode if the Apple device supports it. This upgrade lets accessories support a wider range of Apple devices while still taking advantage of USB Host Mode whenever possible.

Accessories must meet all the following requirements to be able to upgrade from UART to USB Host Mode:

- The accessory must be designed to put the Apple device into USB Host Mode on startup by means of hardware, as specified in the section "Using an Apple Device as a USB Host" in *MFi Accessory Hardware Specification*.

- The accessory must not support the 3G iPod.
- The accessory must communicate on only one transport (UART or USB) at any given time.
- The accessory's hardware must disconnect its USB D+/D- lines, or place them in a tristate condition, so they float when the USB transport is not active.

The accessory must use the process specified below to determine whether the Apple device supports USB Host Mode and to update it to that mode. It must execute this process every time it detects a transition on the Accessory Power pin of the 30-pin connector from low to high state.

1. Immediately upon detecting Accessory Power, the accessory must tristate (float) its USB D+/D- signals (not actively drive them high or low).
2. The accessory must begin its identification process over UART transport by sending a `StartIDPS` command. It must then wait for the Apple device to respond with a General lingo `iPodAck` command.
    - If the Apple device responds with an `iPodAck` error status, IDPS and USB host mode are not supported; the accessory must use only the UART transport for iAP communications. The accessory must now exit this update process.
    - If the Apple device sends an `iPodAck` success status in response to the `StartIDPS` command, the IDPS process is supported. The accessory must proceed to Step 3.
3. The accessory must send a General lingo `GetiPodOptionsForLingo` command to the Apple device, passing a `LingoID` of 0x00 (General lingo). The accessory must then wait for the `RetiPodOptionsForLingo` response.
    - If the USB host mode support bit 27 (listed in Table 3-132 (page 192)) returned by `RetiPodOptionsForLingo` is 0, the Apple device does not support USB host mode; the accessory must continue to communicate over UART transport. The accessory must now exit this update process.
    - If the USB host mode support bit 27 returned by `RetiPodOptionsForLingo` is 1, the Apple accessory supports hardware switching into USB Host Mode. The accessory must proceed to Step 4.
4. The accessory must cancel the IDPS process that it began on the UART transport, by sending a General lingo `EndIDPS` command that passes an `accEndIDPSStatus` value of 3 (see Table 3-97 (page 173)). After receiving an `IDPSStatus` response from the Apple device, the accessory must cease communication over UART transport.
5. The accessory must enable its USB D+/D- lines at the 30-pin connector and set D+ active to indicate that it is a full-speed or high-speed USB device.
6. The accessory must wait for the Apple device to enumerate the accessory as a USB full-speed or high-speed USB device, following the enumeration process specified in the USB 2.0 standard.

2.2.2.3 Packet Formats for iAP Commands

All iAP command packets transferred over the USB IN and OUT pipes must follow the formats specified in "Command Packets" (page 109), except that the sync byte (byte 0) of each packet is unnecessary and should be omitted. If the accessory receives a value of 0xFF as the first byte of a packet, it must ignore that byte.

2.2.2.4 Data Transfers With an Apple Device

When the Apple device enters USB Host Mode, it detects and enable the accessory's USB iAP interface, which consists of an interrupt IN pipe, bulk IN pipe, and bulk OUT pipe. See *MFi Accessory Hardware Specification* for details.

While it is in USB Host mode, the Apple device continuously polls the USB interrupt IN pipe, waiting to receive a zero-length packet (ZLP) data transfer from the accessory. When it has iAP commands or data ready to be read on the bulk IN pipe, the accessory must signal the Apple device by sending a single ZLP data transfer on the interrupt IN pipe. The Apple device then reads the bulk IN pipe repeatedly, requesting the maximum bulk IN USB packet size, until the accessory returns less data than the maximum packet size.

Note: The sync byte (byte 0) of all iAP command packets transferred over USB is unnecessary and should be omitted. If the accessory receives a value of 0xFF for byte 0, it must ignore that byte.

The accessory does not need to send another ZLP data transfer on the interrupt IN pipe as long as it continues to return the maximum USB packet size in response to each bulk IN read. After the accessory returns less data than the maximum packet size, it must send another ZLP data transfer at the next interrupt IN poll interval if it has more data to be read on the bulk IN pipe. If the accessory does not have any bulk IN data to send to the Apple device, it should respond to the USB interrupt IN pipe read by sending a USB NAK packet.

The Apple device will continue to poll the interrupt IN pipe, even while a bulk IN read is currently in progress. If the accessory has no new data transfer to send after the current transfer completes, it should respond to interrupt IN pipe reads by sending USB NAK packets. If the accessory wants to begin a new data transfer after the current transfer completes, it must respond to the next interrupt IN pipe read with by sending a ZLP data transfer. When the Apple device receives a ZLP data transfer on its interrupt IN pipe while a bulk IN read is currently in progress, it will queue up a new bulk IN read internally, to be started immediately after the current bulk IN transfer finishes.

If the Apple device's internal buffer pool is full, it may temporarily stop reading the bulk IN pipe. This is normal behavior; the Apple device will resume reading the bulk IN pipe (as needed) after one or more internal buffers become available. If bulk IN reads stop, even though the maximum packet size was returned with the last read, the accessory must not send more than one additional ZLP data transfer to signal that its bulk IN pipe should be read.

In USB Host mode, the Apple device sends data to the accessory using the bulk OUT pipe. The accessory must return a USB ACK packet if the write operation is successful or return a USB NAK packet if it is not ready to accept data. If the accessory repeatedly returns a USB NAK packet for more than 1 second, the write operation will time out and information from the Apple device may be lost.

Note: The accessory must be able to handle bulk IN and bulk OUT data transfers that occur concurrently.

2.2.2.4.1 Sample Data Transfer to an Apple device

Table 2-2 (page 92) illustrates a typical USB data transfer sequence from the accessory to an Apple device in USB Host mode.

Table 2-2 Sample USB data transfer from accessory to host

Step	Apple device	Accessory
1	Poll the USB interrupt IN pipe.	
2		Return a USB NAK packet in response to the interrupt IN pipe read because the accessory has no data to send to the Apple device.
Continue steps 1 and 2 until the accessory has data to send to the Apple device. When the accessory has data to send, it goes to step 3 in response to the next interrupt IN pipe read.		
3		Return a ZLP data transfer on the interrupt IN pipe in response to the interrupt IN pipe read.
4	Read the USB bulk IN pipe.	
5		Return a packet of the maximum USB packet size on the USB bulk IN pipe in response to the Apple device's bulk IN pipe read.
Continue steps 4 and 5 until the accessory will be returning a packet of less than the maximum USB packet size in response to the next bulk IN pipe read. When the accessory has less than the maximum USB packet size of data to return, it must go to step 6 in response to the next bulk IN pipe read.		
6		Return a packet of less than the maximum USB packet size (or a ZLP) on the USB bulk IN pipe. This tells the Apple device that the accessory is completing the current bulk IN pipe data transfer.
To initiate another bulk IN pipe data transfer immediately, the accessory must send a new ZLP transfer on the interrupt IN pipe (or have sent one while the current transfer was in process), effectively returning to step 3. Otherwise, it must return to step 1.		

2.2.2.5 Optimizing Data Transfers

To achieve maximum data transfer rates to the Apple device using iAP data transfers, accessory developers should observe these hints and cautions:

- After the accessory sends a ZLP data transfer to the Apple device on the interrupt IN pipe (to initiate a data transfer on the bulk IN pipe), the accessory should respond to each bulk IN read by returning a packet of the maximum USB packet size. The accessory should continue doing this until the last USB packet, when it may return a partial or empty packet.
- The accessory should avoid sending data payloads smaller than the maximum USB packet payload size, because afterward it must send another ZLP data transfer on the interrupt IN pipe to reinitiate the bulk IN read process. The latency time between a ZLP data transfer on the interrupt IN pipe and a bulk IN read can be as long as the polling interval, introducing additional data transfer delays and lowering throughput.
- If the bulk IN data packet being returned is exactly the size of the USB packet payload, the Apple device will go on to read the next USB packet on the bulk IN pipe. The accessory must respond to that read with a ZLP data transfer to ensure that the bulk IN pipe read process terminates properly.

- The iAP packets sent over USB may cross USB packet boundaries, and USB packets may cross iAP packet boundaries. If possible, multiple iAP packets should be concatenated or combined to fill each USB packet up to the maximum USB packet payload size, thereby minimizing the number of transitions between the interrupt IN and bulk IN pipes. Each of these transitions introduces an interrupt IN pipe polling interval latency, which should be avoided because it slows the overall data transfer throughput.
- When transferring data in iAP packets, accessories should send the largest iAP packets possible. This permits more data to be sent to the Apple device before a response is required (such as an iAP `iPodAck` command). Sending small packets and waiting for the Apple device to respond introduces unnecessary delays and lowers data throughput.
- Accessories must not send extra interrupt IN pipe ZLPs beyond those required to signal the Apple device to read the bulk IN pipe. If surplus ZLPs are sent, they may cause the Apple device to poll the bulk IN pipe unnecessarily, thereby reducing the USB bus bandwidth available for other purposes. When the accessory sends surplus ZLPs without available data to read on the bulk IN pipe, the Apple device applies a bulk IN read timeout of 1 second. The bulk IN read transfer from the accessory must finish before the timeout. If the transfer does not finish before the timeout, the accessory must send a new ZLP to restart the Apple device's read of the bulk IN pipe.

2.2.3 USB Device Mode

When using iAP over USB Device Mode, the accessory's USB Host component must detect an attached Apple device and select its iUI configuration. After completing this process, the accessory must identify itself as specified in "Identification" (page 93).

2.2.4 Bluetooth Transport

To use iAP over Bluetooth, the accessory must first establish an RFCOMM protocol session with the Apple device. While the session is active, the host may send General lingo iAP commands to the Apple device through the RFCOMM channel. After establishing the session, the accessory must identify itself as specified in "Identification" (page 93).

2.3 Identification

Accessories use the Identify Device Preferences and Settings (IDPS) process to identify themselves to an Apple device. After completing the IDPS process, accessories perform the process specified in "Authentication" (page 103).

IMPORTANT: To ensure compatibility with future firmware, the designs of new accessories must use the IDPS process, except for accessories that only supply power to the Apple device. Existing accessories may continue to send the iAP General lingo command 0x13, `IdentifyDeviceLingoes`, identifying the lingoes they support and requesting immediate authentication. Note that IDPS requires accessory Authentication 2.0, as specified in "Authentication" (page 103), and the use of transaction IDs throughout each communication session, as specified in "Transaction IDs" (page 110).

2.3.1 IDPS Commands

The General lingo IDPS commands that support accessory identification are listed below. All commands are protocol version 1.09 and require authentication 2.0.

Table 2-3 IDPS General lingo commands

Cmd ID	Command name	Direction	Parameters:bytes
0x38	`StartIDPS`	Acc to Dev	`{transID:2}`
0x11	`RequestTransport-MaxPayloadSize`	Acc to Dev	`{transID:2}`
0x12	`ReturnTransport-MaxPayloadSize`	Dev to Acc	`{transID:2, maxSize:2}`
0x39	`SetFIDTokenValues`	Acc to Dev	`{transID:2, numFIDTokenValues:1, FIDTokenValues:<var>}`
0x3A	`AckFIDTokenValues`	Dev to Acc	`{transID:2, numFIDTokenValueACKs:1, FIDTokenValueACKs:<var>}`
0x3B	`EndIDPS`	Acc to Dev	`{transID:2, accIDPSStatus:1}`
0x3C	`IDPSStatus`	Dev to Acc	`{transID:2, status:1}`

Note: The `transID` parameters in the foregoing table are described in "Transaction IDs" (page 110).

2.3.2 The IDPS Process

The IDPS process begins when an accessory sends a `StartIDPS` command to an Apple device (see "Command 0x38: StartIDPS" (page 160)). If the two are newly connected, it must be the first command the accessory sends. If the Apple device refuses `StartIDPS` by returning a General lingo `iPodAck` command with a status of 0x04 (Bad Parameter), the accessory must assume it is connected to an Apple device that doesn't support the IDPS process. In this case, the accessory must send the Apple device an `IdentifyDeviceLingoes` command within 800 ms, requesting authentication.

IMPORTANT: If the accessory has access to Accessory Power through the 30-pin connector, it must always monitor that output from the Apple device (pin 13). If Accessory Power goes low, even momentarily, the accessory must restart the IDPS process after it goes high. After Accessory Power goes high the accessory must wait at least 80 ms before starting the IDPS process or sending any other iAP commands.

In addition to identifying itself at plug-in, an accessory may need to reidentify during an iAP session if the Apple device so requests. If the accessory receives "Command 0x00: RequestIdentify" (page 124), it must send a `StartIDPS` command and repeat the IDPS and authentication processes. USB Device Mode accessories do not have to reconfigure iUI when responding to the `RequestIdentify` command from the Apple device.

Note: An accessory that is newly connected to a sleeping or hibernating Apple device cannot expect an immediate response. If it uses wired transport, the accessory should wait for the Apple device to turn on Accessory Power. If it uses Bluetooth or is unsuccessful the first time, it should retry the `StartIDPS` command every 500 ms until it gets a response.

During the IDPS process, the Apple device accepts only the following General lingo commands from the accessory:

- 0x11, `RequestTransportMaxPayloadSize` to determine the maximum payload size of the current iAP transport
- 0x4B, `GetiPodOptionsForLingo` to determine the options the Apple device supports for a specific lingo
- 0x39, `SetFIDTokenValues` to identify the accessory to the Apple device
- 0x3B, `EndIDPS` to end the IDPS process

IMPORTANT: After an Apple device has successfully acknowledged an accessory's `StartIDPS` command, all subsequent iAP command packets must include transaction IDs, regardless of lingo, as specified in "Transaction IDs" (page 110).

The IDPS process lets the Apple device receive preference information from the accessory during the initial handshake sequence. Unlike the process using `IdentifyDeviceLingoes`, the audio/video routing and other Apple device settings are established before authentication begins. Once the IDPS process begins, the Apple device will not enable preferences such as line-out and video-out by default; therefore it is up to the accessory to explicitly enable all desired preferences.

IMPORTANT: To avoid the Apple device displaying a compatibility warning, every accessory must start its authentication process within 2 seconds after the Apple device has detected it. Failure to do this reliably is not acceptable. For the conditions under which an Apple device detects an accessory, see *MFi Accessory Hardware Specification*. Also, the total times that it takes the accessory to respond to Apple device commands during the IDPS process must not add up to more than 3 seconds, excluding the Apple device's response times.

Note: During the IDPS process, the Apple device identifies all connected accessories as uniquely as possible. Each accessory should enable all the preferences that it requires in the Apple device during its IDPS process. An accessory may change those preferences at any later time by sending a `SetiPodPreferences` command.

2.3.2.1 Starting IDPS

An accessory sends the `StartIDPS` command to start the Identify Device Preferences and Settings (IDPS) process. When the Apple device sends a General lingo `iPodAck` command with a status of Success (0x00) in response to `StartIDPS`, it enters the IDPS process described in "The IDPS Process" (page 94). If the accessory sends `StartIDPS` again, while the Apple device is in the IDPS process, the Apple device restarts the IDPS process.

After starting the IDPS process, the accessory must send a General lingo `RequestTransportMaxPayloadSize` command. The Apple device responds with a `ReturnTransportMaxPayloadSize` command, passing the maximum payload size allowed by the current iAP transport (UART, USB Device mode, USB Host mode, or Bluetooth). The accessory must store this value to help it send subsequent iAP commands successfully.

If at any time the Apple device sends a General lingo `iPodAck` command with a status of Maximum Connections (0x15) the accessory must not continue with IDPS or any other identification process, because the Apple device has already reached its maximum allowed number of accessory connections.

2.3.2.2 Completing IDPS

Normally, the accessory must start and complete the IDPS process (defined as sending `EndIDPS` successfully) within a total time of 3 seconds after the rising edge of power from the Apple device on pin 13 (Accessory Power) of the 30-pin connector. The accessory may continue the IDPS process only after it has received an `iPodAck` response to `StartIDPS` whose transaction ID matches that of the accessory's most recent `StartIDPS` command. Each `StartIDPS` command renders previous `StartIDPS` commands invalid, even if they are subsequently acknowledged.

After the IDPS process has finished, the Apple device sends the accessory an `IDPSStatus` command. If this command indicates an unsuccessful IDPS process, the accessory can retry `StartIDPS` within the 3-second total time limit. If the IDPS process times out, the accessory won't be able to retry IDPS unless it is detached and re-attached, or the Apple device sends a `RequestIdentify` command.

If the accessory is in the IDPS process and cannot complete it successfully, it must send `EndIDPS` with a status value of 0x02 before either retrying IDPS or sending `IdentifyDeviceLingoes`. If the accessory does this within 3 seconds of sending `StartIDPS`, the Apple device will send an `IDPSStatus` of 0x04 and will let the accessory retry the IDPS process or send `IdentifyDeviceLingoes`. If the accessory fails to do this within 3 seconds of sending `StartIDPS`, the Apple device will not let the accessory retry IDPS and will send an `IDPSStatus` of 0x05 in response to the accessory's `EndIDPS` command. After that, the Apple device will accept only `IdentifyDeviceLingoes`.

If an accessory has already completed IDPS successfully, inadvertently sending a `StartIDPS` or `IdentifyDeviceLingoes` command resets its authentication and identification states. The accessory must repeat the IDPS and authentication processes.

When an Apple device transitions from sleep to power on, it sends a `RequestIdentify` command to the accessory. The accessory must respond with `StartIDPS` and go through the IDPS process. After an Apple device transitions from hibernation and the accessory detects current from the Apple device on the Accessory Power line of the 30-pin connector, the accessory must wait at least 80 ms, send `StartIDPS`, and complete the IDPS process.

When an Apple device exits its sleep state or is connected to an accessory for the first time after it boots, the accessory might notice a delay in its response to `StartIDPS`. In that case, the accessory must complete IDPS within 3 seconds after receiving a successful `iPodAck` in response to its `StartIDPS`.

2.3.3 Determining Apple Device Capabilities

During the IDPS process, the accessory should determine the capabilities of its attached Apple device, so that the accessory does not try to declare and use features that the device cannot handle. This is done by calling `GetiPodOptionsForLingo` after `StartIDPS`. The typical procedure is the following, which includes alternatives for older Apple devices that do not support `StartIDPS` or `GetiPodOptionsForLingo`:

1. Send a `StartIDPS` command.
2. if the `iPodAck` response to `StartIDPS` passes a failure status, jump to the procedure listed in "Apple Devices That Require IdentifyDeviceLingoes" (page 97).
3. To test `GetiPodOptionsForLingo` command support, send a `GetiPodOptionsForLingo` command for the General lingo (0x00) and wait for a response from the Apple device.
4. If the Apple device returns `RetiPodOptionsForLingo`, the accessory can use `GetiPodOptionsForLingo` for the remainder of its capabilities queries; go to Step 5. If the accessory receives an `iPodAck` command passing a failure status, jump to the procedure listed in "Apple Devices That Require IdentifyDeviceLingoes" (page 97).
5. Test the bits in the value returned by `RetiPodOptionsForLingo` to determine the Apple device's general capabilities and features.
6. Determine other Apple device capabilities and features by repeatedly sending `GetiPodOptionsForLingo` commands for other lingoes of interest to the accessory. In each case, if the response is an `iPodAck` command passing a failure status, the Apple device does not support the lingo; otherwise, the accessory can test the bits in the value returned by `RetiPodOptionsForLingo` to determine the Apple device's capabilities for that lingo.
7. Finish the identification process. Send a `SetFIDTokenValues` command (as illustrated in "Sample IDPS Command Sequences" (page 98)), using the results of the foregoing capability determination to set appropriate token values, such as the accessory's preferences within each capability.

2.3.3.1 Apple Devices That Require IdentifyDeviceLingoes

The following is a typical procedure for determining the capabilities of older Apple devices that do not support `StartIDPS` or `GetiPodOptionsForLingo`:

1. Send `GetiPodOptions` and test bits to determine desired iPod feature support, such as line output and video output. If the response is an `iPodAck` command passing a failure status, the Apple device may be a 3G iPod; see "The 3G iPod" (page 527).

2. Determine other Apple device capabilities and features by repeatedly sending `RequestLingoProtocolVersion` commands for other lingoes of interest to the accessory. In each case, if the response is an `iPodAck` command passing a failure status, the Apple device does not support the lingo; otherwise, the lingo version information returned by `ReturnLingoProtocolVersion` can help the accessory determine the Apple device's capabilities for that lingo.

3. Proceed with the identification process by sending an `IdentifyDeviceLingoes` command requesting authentication; see "Accessory Identification With Non-IDPS Apple Devices" (page 553).

2.3.4 Sample IDPS Command Sequences

This section lists two typical command sequences using IDPS:

- Table 2-4 (page 98) illustrates the basic IDPS command sequence.
- Table 2-5 (page 100) illustrates an IDPS command sequence followed by authentication and accessory support for Digital Audio (Lingo 0x0A).

Table 2-4 IDPS process up to authentication

Step	Accessory command	Apple device command	Comment
1	`StartIDPS`		no params
2		`iPodAck`	acknowledging command 'General Lingo::StartIDPS'
If the `iPodAck` reply to `StartIDPS` returns a status of 0x00, the accessory proceeds to Step 3. Otherwise, ■ If the accessory receives no `iPodAck` command within 500 ms, it may retry Step 1 at 500 ms intervals as long as Apple Device Detect (pin 30 of the 30-pin connector) is low and Accessory Power (pin 13) is high. Accessories such as Bluetooth accessories or USB accessories without a 191 kΩ ID resistor, which don't monitor these signals, may retry Step 1 without checking them. ■ If the accessory receives an `iPodAck` status of 0x04 (Bad Parameter), the Apple device does not support IDPS. The accessory must not proceed to Step 3. Instead, it must perform either of these two alternative actions: ❏ If the accessory's function requires IDPS support, within 800 ms it must send an `IdentifyDeviceLingoes` command with all parameters set to 0xFF. This makes the Apple device display an "Accessory not supported" message. ❏ If the accessory does not require IDPS support, the accessory must send an `IdentifyDeviceLingoes` command with its lingo mask set for only the General lingo, with no options, and with a Device ID of 0x00. See "Canceling a Current Authentication Process With IdentifyDeviceLingoes" (page 135). A sample command sequence is listed in Table C-48 (page 553). ■ If the accessory receives any other nonzero `iPodAck` status, it must not send any more iAP commands until it has been disconnected. Upon reconnection it must restart identification at Step 1.			
3	`RequestTransport-MaxPayloadSize`		requesting transport maximum payload size

Step	Accessory command	Apple device command	Comment
4		ReturnTransport-MaxPayloadSize	returning transport maximum payload size
5	SetFIDTokenValues		setting 11 FID tokens ; IdentifyToken = (lingoes: 0/2/3/4|options 0x00000002|device ID 0x00000200); AccCapsToken = 0x0000000000000205; AccInfoToken = Acc name (Test-Accessory); AccInfoToken = Acc FW version (v1.1.1); AccInfoToken = Acc HW version (v2.2.2); AccInfoToken = Acc manufacturer (Apple; Inc.); AccInfoToken = Acc model number (Test-Model); EAProtocolToken = 1 (com.Apple.ProtocolMain); EAProtocolToken = 2 (com.Apple.ProtocolAlt); iPodPreferenceToken = (setting 'off' for preference class 'video out setting' with restore on exit 'not selected'); iPodPreferenceToken = (setting 'used' for preference class 'line out usage' with restore on exit 'not selected')
6		AckFIDTokenValues	11 ACKs for FID tokens ; IdentifyToken = accepted; AccCapsToken = accepted; AccInfoToken = (Acc name) accepted; AccInfoToken = (Acc FW version) accepted; AccInfoToken = (Acc HW version) accepted; AccInfoToken = (Acc manufacturer) accepted; AccInfoToken = (Acc model number) accepted; EAProtocolToken = (1) accepted; EAProtocolToken = (2) accepted; iPodPreferenceToken = (video out setting) accepted; iPodPreferenceToken = (line out usage) accepted
7	EndIDPS		status 'finished with IDPS; proceed to authentication'
8		IDPSStatus	status 'ready for auth'
9		GetAccessory-AuthenticationInfo	no params

Table 2-5 IDPS process with authentication and Digital Audio lingo

Step	Accessory command	Apple device command	Comment
1	`StartIDPS`		no params
2		`iPodAck`	acknowledging command 'General Lingo::StartIDPS'
If the `iPodAck` reply to `StartIDPS` returns a status of 0x00, the accessory proceeds to Step 3. Otherwise, ■ If the accessory receives no `iPodAck` command within 500 ms, it may retry Step 1 at 500 ms intervals as long as Apple Device Detect (pin 30 of the 30-pin connector) is low and Accessory Power (pin 13) is high. Accessories such as Bluetooth accessories or USB accessories without a 191 kΩ ID resistor, which don't monitor these signals, may retry Step 1 without checking them. ■ If the accessory receives an `iPodAck` status of 0x04 (Bad Parameter), the Apple device does not support IDPS. The accessory must not proceed to Step 3. Instead, it must perform either of these two alternative actions: ❏ If the accessory's function requires IDPS support, within 800 ms it must send an `IdentifyDeviceLingoes` command with all parameters set to 0xFF. This makes the Apple device display an "Accessory not supported" message. ❏ If the accessory does not require IDPS support, the accessory must send an `IdentifyDeviceLingoes` command with its lingo mask set for only the General lingo, with no options, and with a Device ID of 0x00. See "Canceling a Current Authentication Process With IdentifyDeviceLingoes" (page 135). A sample command sequence is listed in Table C-48 (page 553). ■ If the accessory receives any other nonzero `iPodAck` status, it must not send any more iAP commands until it has been disconnected. Upon reconnection it must restart identification at Step 1.			
3	`RequestTransport-MaxPayloadSize`		requesting transport maximum payload size
4		`ReturnTransport-MaxPayloadSize`	returning transport maximum payload size

Step	Accessory command	Apple device command	Comment
5	SetFIDTokenValues		setting 11 FID tokens ; IdentifyToken = (lingoes: 0/2/3/4/0x0A|options 0x00000002|device ID 0x00000200); AccCapsToken = 0x0000000000000215; AccInfoToken = Acc name (Test-Accessory); AccInfoToken = Acc FW version (v1.1.1); AccInfoToken = Acc HW version (v2.2.2); AccInfoToken = Acc manufacturer (Apple; Inc.); AccInfoToken = Acc model number (Test-Model); EAProtocolToken = 1 (com.Apple.ProtocolMain); EAProtocolToken = 2 (com.Apple.ProtocolAlt); iPodPreferenceToken = (setting 'off' for preference class 'video out setting' with restore on exit 'not selected'); iPodPreferenceToken = (setting 'used' for preference class 'line out usage' with restore on exit 'not selected')
6		AckFIDTokenValues	11 ACKs for FID tokens ; IdentifyToken = accepted; AccCapsToken = accepted; AccInfoToken = (Acc name) accepted; AccInfoToken = (Acc FW version) accepted; AccInfoToken = (Acc HW version) accepted; AccInfoToken = (Acc manufacturer) accepted; AccInfoToken = (Acc model number) accepted; EAProtocolToken = (1) accepted; EAProtocolToken = (2) accepted; iPodPreferenceToken = (video out setting) accepted; iPodPreferenceToken = (line out usage) accepted
7	EndIDPS		status 'finished with IDPS; proceed to authentication'
8		IDPSStatus	status 'ready for auth'
9		GetAccessory-AuthenticationInfo	no params

Step	Accessory command	Apple device command	Comment
10	RetAccessory-AuthenticationInfo		returning version of authentication 2.0 and X.509 certificate; see Table 3-28 (page 137)
11		AckAccessory-AuthenticationInfo	acknowledging 'auth info supported'
12		GetAccSampleRateCaps	no params
13		GetAccessory-Authentication-Signature	offering challenge for the accessory to sign and return; see Table 3-30 (page 138)
14	RetAccSampleRateCaps		returning sample rates '8000|11025|12000|16000|22050|24000|32000|44100|48000'
15	RetAccessory-Authentication-Signature		returning digital signature; see Table 3-31 (page 139)
16		TrackNewAudio-Attributes	sample rate 44100; Sound Check value 0; track volume adjustment 0
17	AccAck		acknowledging 'TrackNewAudioAttributes'
18		AckAccessory-AuthenticationStatus	acknowledging authentication status 'Success (OK)'
19		TrackNewAudio-Attributes	sample rate 44100; Sound Check value 0; track volume adjustment 0
20	AccAck		acknowledging 'TrackNewAudioAttributes'
21		TrackNewAudio-Attributes	sample rate 44100; Sound Check value 0; track volume adjustment 0
22	AccAck		acknowledging 'TrackNewAudioAttributes'

2.4 Authentication

The authentication process follows the IDPS process. It is a mechanism used by an Apple device to verify that an attached accessory is an authorized accessory and by an accessory to authenticate the Apple device, if desired. Certain functionality on the Apple device is accessible only after an accessory has been authenticated as an authorized accessory. This functionality is summarized in Table 2-7 (page 104) and includes the use of any accessory lingo command over the USB or BT transport.

The limited set of General lingo commands listed in Table 2-7 (page 104) as requiring no authentication are free over USB and BT. After the accessory identification process, authentication may be independently initiated from either the Apple device or the accessory. In the identify and authentication sequences, the accessory should check the `iPodAck` status for all commands. If a failure status is returned, the process has failed and should be retried and/or abandoned.

Note: An accessory must not retry the IDPS process until at least 500 ms has passed.

IMPORTANT: To participate in the authentication process, an accessory must contain an authentication coprocessor provided by Apple.

2.4.1 Levels of Accessory Authentication

iAP supports three levels of security in the process by which an Apple device may authenticate an accessory connected to it:

- **None.** Older Apple device models through the iPod mini do not support authentication. They operate freely with simple accessories that do not require authentication, but they cannot work with newer or more sophisticated accessories whose range of functionality requires authentication. For details, see "Functional Description" in *MFi Accessory Hardware Specification*.
- **Authentication 1.0.** Version 1.0 authentication is based on public and private keys, where each accessory has a private key and every Apple device has the associated public key. The accessory authentication process is a part of the iAP General lingo (0x00) command protocol and is controlled by the Apple device's internal software. Accessories identify themselves to the Apple device as speaking specific lingoes and supporting authentication. iAP queries the accessory's authentication information, sends a challenge to the accessory, and verifies that the accessory responds to the challenge correctly.
- **Authentication 2.0.** Version 2.0 authentication is based on standard X.509 Version 3 certificates. Information about this standard can be found at the IETF website: http://tools.ietf.org/html/rfc3280. Each certificate is generated and signed by a recognized certificate authority (CA) and has a unique serial number. Authentication 2.0 uses certificate classes to identify the type of accessory being authenticated, as shown in Table 2-6 (page 104).

IMPORTANT: Current Apple devices support Authentication 1.0, as shown in "Authentication 1.0 Sample Command Sequence" (page 552), only as a legacy technology, to make them compatible with existing accessories. All new accessory designs must use Authentication 2.0.

Table 2-6 Apple device authentication coprocessor classes

iPod Class ID	iPhone Class ID	Description
1	4	Automotive use only; enables all authenticated features.
2	5	Nonautomotive use only; enables all authenticated features except the Digital Audio lingo. **Deprecated; do not use in new products.**
3	6	Nonautomotive use only; enables all authenticated features.

2.4.2 Authentication Requirements

Because accessory authentication can take a significant amount of time (up to 75 seconds for Authentication 2.0A), it is performed as a background process. The process starts when the Apple device sends a `GetAccessoryAuthenticationInfo` command to the accessory. The accessory must respond by transmitting its entire `RetAccessoryAuthenticationInfo` command within 1.00 seconds for Authentication 1.0 or 2.00 seconds for Authentication 2.0, including the transmission of all sections of its certificate. When the Apple device transmits a `GetAccessoryAuthenticationSignature` command, the accessory must transmit its `RetAccessoryAuthenticationSignature` response within 7.00 seconds for Authentication 1.0, 75.00 seconds for Authentication 2.0A, or 2.00 seconds for Authentication 2.0B and 2.0C.

All lingo-authenticated commands and features are available to accessories once the Apple device has sent the accessory an `AckAccessoryAuthenticationInfo` command with success status (0x00). This provisional authentication lasts until the Apple device sends a `AckAccessoryAuthenticationStatus` command indicating that authentication has finished. With accessories attached through the 30-pin connector, the lingo command availability is revoked if authentication process fails, the accessory reidentifies without authentication, the Apple device sleeps or powers on, or the accessory is detached from the Apple device. The same is true of accessories connected through Bluetooth, except that they remain authenticated while the Apple device sleeps. The Apple device starts both a timer and a retry counter to guarantee that the authentication process will conclude.

IMPORTANT: An accessory may send only the commands allowed for the combinations of transports and authentication states listed in Table 2-7 (page 104). However, every accessory must send commands that respond directly to commands from an Apple device regardless of transport and authentication state.

Table 2-7 Authentication requirements for iAP commands

Lingoes	UART transport	USB transport	BT transport
Lingo 0x00: General	No (0x03-0x06)	Yes (0x03-0x06)	

Lingoes	UART transport	USB transport	BT transport
	No (0x00-0x02, 0x07-0x16[1], 0x23-0x25, 0x27-0x28, 0x38-0x3C, 0x46-0x4C[1], 0x67-0x68) Yes (0x17-0x1F, 0x29-0x2B[2], 0x35-0x37, 0x3F-0x43[2], 0x4D-0x51, 0x54, 0x64-0x66, 0x69-0x6A)		
Lingo 0x01: Microphone	**Deprecated**		
Lingo 0x02: Simple Remote	No (0x00) Yes (0x01, 0x03–0x04, 0x0B-0x1A, 0x81)	Yes	Yes[3]
Lingo 0x03: Display Remote	No (0x00–0x07, 0x1A–0x1E) Yes (0x08–0x19, 0x1F–0x22)	Yes	Yes[3]
Lingo 0x04: Extended Interface	No (0x00–0x3A) Yes (0x3B-0x45, 0x47-0x49)	Yes	Yes[3]
Lingo 0x05: Accessory Power	**Deprecated**		
Lingo 0x06: USB Host Mode	Yes	N/A	N/A
Lingo 0x07: RF Tuner	Yes	Yes	N/A
Lingo 0x08: Accessory Equalizer	Yes	Yes	N/A
Lingo 0x09: Sports	Yes	Yes	Yes
Lingo 0x0A: Digital Audio	Yes	Yes	N/A
Lingo 0x0B: Reserved	N/A	N/A	N/A
Lingo 0x0C: Storage	Yes	Yes	Yes
Lingo 0x0D: iPod Out	Yes	Yes	No
Lingo 0x0E: Location	Yes	Yes	Yes
Lingoes 0x0F–0x1F: Reserved	N/A	N/A	N/A

[1] Apple devices will accept the General lingo commands 0x11 (`RequestTransportMaxPayloadSize`) and 0x4B (`GetiPodOptionsForLingo`) from an accessory during the IDPS process, before authentication. After the IDPS process these commands must be authenticated.

[2] Preference commands (0x29-0x2B) require authentication on all Apple devices except the 5G iPod; however, getting or setting the line-out preference class (0x03) does not require authentication. Commands 0x42 (`AccessoryDataTransfer`) and 0x43 (`iPodDataTransfer`) are not available until authentication has finished.

[3] Bluetooth accessories that support the Simple Remote, Display Remote, or Extended Interface lingoes must have their own volume controls and not use iAP volume control commands. They must also implement Extended Interface playback status notifications, as specified in "Playback Status Notifications" (page 59).

2.4.3 Apple Device Authentication of the Accessory

The process of authenticating the accessory is initiated by the Apple device, based on the settings made in the IDPS process. The authentication options also allow an accessory to request authentication of an Apple device, as described in "Accessory Authentication of the Apple Device" (page 108).

IMPORTANT: The accessory must send the Apple device a `StartIDPS` command before it tries to send any other iAP commands.

Table 2-8 (page 106) summarizes the authentication process, using Authentication 2.0. This is the authentication process that all new accessories should use. Steps 4 and 5 are necessary only if the accessory is unable to use the IDPS process and is initializing using `IdentifyDeviceLingoes`.

Table 2-8 Command traffic for accessory authentication

Step	Action or command	Direction	Comments
1	`GetAccessory-AuthenticationInfo` (0x14)	Dev to Acc	The Apple device requests accessory authentication information and starts its timeout timer.
2	`RetAccessory-AuthenticationInfo` (0x15)	Acc to Dev	The accessory returns its major and minor authentication version and its X.509 public certificate (see Note 1, below).
3	`AckAccessory-AuthenticationInfo` (0x16)	Dev to Acc	The Apple device assembles and checks the accessory's X.509 certificate. If it does not support the authentication version, or if the certificate check fails, the Apple device sends a value of 0x08 to the accessory (see Note 2, below).
All lingo-authenticated commands are enabled after the authentication version is validated, the lingoes requested by the accessory are checked against the lingoes allowed by the X.509 certificate, and the certificate has been verified (see Note 3, below).			
4 (non-IDPS only)	`GetAccessoryInfo` (0x27)	Dev to Acc	The Apple device queries the accessory for information about it.
5 (non-IDPS only)	`RetAccessoryInfo` (0x28)	Acc to Dev	The accessory returns information about its identity and capabilities.
6	`GetAccessory-Authentication-Signature` (0x17)	Dev to Acc	The Apple device sends a 20-byte random challenge to the accessory and asks it to calculate a digital signature.
7	`RetAccessory-Authentication-Signature` (0x18)	Acc to Dev	The accessory returns its digital signature to the Apple device within 2 seconds (Authentication 2.0B) or 75 seconds (Authentication 2.0A).

Step	Action or command	Direction	Comments
8	AckAccessory-AuthenticationStatus (0x19)	Dev to Acc	The Apple device verifies the signature, using the public key contained in the accessory's X.509 certificate, and returns the status of signature verification.

Notes:

1. The maximum number of certificate bytes that may be sent in one command is determined by calling the command RequestTransportMaxPayloadSize. If the X.509 certificate is larger than this number, the accessory must divide it into sections, each not larger, for reassembly by the Apple device. The Apple device will send a General lingo iPodAck command for each certificate section up to, but not including, the last one. The accessory must wait for the iPodAck command before sending the next certificate section in a RetAccessoryAuthenticationInfo packet.

2. The iPodAck response to each certificate section only acknowledges receipt. When all the sections have been assembled, the Apple device evaluates the whole certificate and sends an AckAccessoryAuthenticationInfo command that states whether or not it is valid.

3. The Apple device parses the X.509 certificate into an allowed lingoes mask. It uses the mask to confirm that the accessory's identified lingoes are allowed by the certificate. If the accessory requests lingoes not allowed by the certificate, authentication fails. The Apple device also verifies that the certificate is valid.

4. If authentication fails, the accessory must wait for at least 5000 ms and then repeat the entire identification and authentication process as described in "Transport Initialization" (page 88). If the Apple device sends a RequestIdentify, GetAccessoryAuthenticationInfo, or GetAccessoryAuthenticationSignature command before the 5000 ms period has elapsed, the accessory must stop waiting and restart the identification or authentication process in response to the Apple device.

The Apple device and the accessory can perform noncritical operations while background authentication is in progress. The command timeout counter and retry counter are not reset until authentication is complete or has failed.

Some lingoes requested by an accessory (such as the General, Accessory Equalizer, and Sports lingoes) cause the Apple device to request accessory information after it sends the AckAccessoryAuthenticationInfo command. Accessories should be able to handle both authentication requests and asynchronous information requests from the Apple device during the authentication process.

With IDPS all Apple device preferences are set before authentication. Accessories that must use IdentifyDeviceLingoes should set any Apple device preferences that do not require authentication within 2 seconds of receiving the iPodAck reply to IdentifyDeviceLingoes, and set Apple device preferences that require authentication within 2 seconds of receiving the AckAccessoryAuthenticationInfo command.

If the Apple device fails to authenticate the accessory at any point in the Authentication 1.0 or 2.0 process, and the retry count is exhausted, the Apple device may display an "Accessory not supported" message to the user.

2.4.4 Accessory Authentication of the Apple Device

The accessory can initiate a process to authenticate the Apple device, any time after it has been authenticated by the Apple device. This process is summarized in Table 2-9 (page 108). See "General Lingo Command Summary" (page 119) for information on the individual commands.

Note: Accessory authentication of the Apple device is currently supported only for Authentication 2.0.

Table 2-9 Accessory authentication of the Apple Device

Step	Action or command	Direction	Comments
1	`GetiPodAuthentication-Info` (0x1A)	Acc to Dev	The accessory requests information to authenticate the Apple device.
2	`RetiPodAuthentication-Info` (0x1B)	Dev to Acc	The Apple device returns its major and minor authentication version and its X.509 public certificate (see Note, below).
3	`AckiPodAuthentication-Info` (0x1C)	Acc to Dev	The accessory assembles and checks the Apple device's X.509 certificate. If the accessory does not support the authentication version, or if the certificate check fails, it returns an error status.
4	`GetiPodAuthentication-Signature` (0x1D)	Acc to Dev	The accessory sends a 20-byte random challenge to the Apple device and asks it to calculate a digital signature.
5	`RetiPodAuthentication-Signature` (0x1E)	Dev to Acc	The Apple device returns its digital signature to the accessory within 75 seconds. The accessory verifies the signature, using the public key contained in the Apple device's X.509 certificate. If verification succeeds, the accessory begins to operate.
6	`AckiPodAuthentication-Status` (0x1F)	Acc to Dev	The accessory returns the status of the digital signature comparison.

Note:

1. The Apple device's X.509 certificate is sent in a PKCS-7 message containing only the certificate, encoded in DER format. See the IETF document RFC 2311, "S/MIME Version 2 Message Specification" for details about using PKCS-7 messages to transfer X.509 certificates. If necessary, the Apple device divides the X.509 certificate into sections, each not larger than the maximum payload size that the accessory specified during the IDPS process, for reassembly by the accessory. If the accessory has not specified a payload size, the Apple device sends 128-byte sections. The accessory must not acknowledge any certificate sections sent by the Apple device until the complete certificate has been received and verified.

2.5 Command Packets

The general format for iAP command packets is shown in Table 2-10 (page 109). The fields shown in the table are discussed in the rest of this section.

Note: Unless otherwise specified, all data units larger than bytes must be transferred in a big-endian order; that is, 32 bits must be sent as bits 31:24, followed by bits 23:16, and so forth. Similarly, 16 bits must be sent as bits 15:8, followed by bits 7:0.

Table 2-10 iAP command packet format

Packet region	Field name	Bytes	Description
Header	`start`	1 or 2	Start of packet; see "Start of Packet Field" (page 109)
	`length`	1 or 3	Length of packet payload; see "Payload Length Field" (page 110)
	`lingoID`	1	Lingo identifier; see Table 4-1 (page 211).
	`commandID`	1 or 2	Command identifier; see command listings in Chapters 3, 4, and 5.
	`transID`	2	Transaction ID; see "Transaction IDs" (page 110)
Payload	For command parameters see command descriptions in Chapters 3, 4, and 5.		
Footer	`checksum`	1	See "Checksum Byte" (page 110).

The commands in the iAP interface are divided into lingoes, which are listed in Table 4-1 (page 211). The General lingo (`lingoID` 0x00) is specified in Chapter 3 and the Extended Interface lingo (`lingoID` 0x04) is specified in Chapter 5. All other lingoes are specified in Chapter 4.

The commands in each lingo are identified by their `commandID` values. The combination of a `lingoID` and a `commandID` uniquely identifies each command in the iAP interface.

Note: The `commandID` values for Extended Interface commands (`lingoID` 0x04) are 2 bytes long; all other `commandID` values are 1 byte long.

2.5.1 Start of Packet Field

The `start` field always contains the value 0x55. With UART transport this byte must be preceded by a 0xFF byte, which facilitates automatic baud rate detection and powers up some Apple devices. For convenience, a `start` field value of 0xFF55 may be used with all transports.

2.5.2 Payload Length Field

The value of the `length` field is the total number of bytes in the packet's `lingoID`, `commandID` and `transID` fields plus the lengths of all command parameters being passed (see Table 2-10 (page 109). It represents the length in bytes of the entire command packet excluding its `start`, `length`, and `checksum` fields.

The `length` field may contain 1 or 3 bytes, depending on the payload length. A 1-byte field can express a payload length of 0x02 to 0xFC (2 to 252) in a single byte. A 3-byte field contains a 0x00 marker byte followed by a 2-byte payload length value from 0x00FD to 0xFFFA (253 to 65529).

Note: Early in the IDPS process the accessory must send "Command 0x11: RequestTransportMaxPayloadSize" (page 132) to discover and store the maximum payload size per packet allowed by the current iAP transport (UART, USB Device mode, USB Host mode, or Bluetooth). If the transport does not support the payload length to be sent in a subsequent command packet, the accessory must use the technique specified in "Multisection Data Transfers" (page 116) to send the command.

2.5.3 Checksum Byte

The `checksum` byte is calculated each time a command is sent. Its value and the values of all the fields in the packet shown in Table 2-10 (page 109), including the payload but excluding the `start` field, must add up to 0x00. The `checksum` value is calculated by adding together the values of the `length`, `lingoID`, `commandID`, and `transID` fields, plus the values of all command parameters being passed, as signed 8-bit values (discarding any signed 8-bit overflow) and then negating that sum to create the signed 8-bit `checksum` byte.

An accessory command packet that has a valid checksum but contains an invalid parameter, invalid command, or other such failure causes the Apple device to respond with an `iPodAck` command containing the appropriate error status.

A packet with an invalid checksum received by an Apple device is presumed to be invalid and is ignored. No `iPodAck` or other command is sent to the accessory in response to the invalid packet.

2.6 Transaction IDs

Once an accessory has started the IDPS process, beginning with and including the `StartIDPS` command, the accessory must include 16-bit transaction IDs in every iAP packet. This section specifies the rules that must be followed when creating or interpreting the values of these IDs.

2.6.1 Using Transaction IDs

The purpose of transaction IDs is to uniquely associate iAP commands with their responses, regardless of the order in which commands and their responses may be transmitted.

Note: Transaction ID fields are part of the packet payload and must be counted when specifying the data lengths of iAP commands.

2.6.1.1 Generating and Returning Transaction IDs

An attached accessory's responsibility for handling transaction IDs can be summarized by the following rules:

- Every time the accessory responds to a command from an Apple device contains a transaction ID, it must include that ID in its response.
- To generate transaction IDs for the commands it sends, the accessory must initialize a 16-bit counter to 0x0000 every time it is connected to an Apple device. It must then use the counter value as the transaction ID for every command it sends and increment the counter afterward. The counter may roll over to 0x0000 after it reaches 0xFFFF.
- If the accessory receives no response to a command that it has sent, it may send another command with the same payload, but the transaction ID must be different.
- The accessory must ignore any response from the Apple device whose transaction ID does not match that of a previous command it has sent.
- The Apple device will ignore any response from the accessory whose transaction ID does not match that of a previous command it has sent.

Note: The Apple device does not increment transaction ID values during authentication. The Accessory must continue to respond to Apple device commands with the transaction ID included in the Apple device's command.

2.6.1.2 Enabling and Disabling Transaction ID Support

Accessories must enable their support for transaction IDs according to the following rules:

- Support for transaction IDs must be enabled upon the rising edge of power from the Apple device on pin 13 (Accessory Power) of the 30-pin connector (see *MFi Accessory Hardware Specification*).
- Support for transaction IDs must be enabled before the accessory sends a `StartIDPS` command.

The accessory must continue to enable its transaction ID support for all iAP commands until it encounters one of the following situations:

- Support for transaction IDs must be disabled upon receipt of a General lingo `iPodAck` command without a transaction ID. Such commands have a payload length value (byte 2) of either 0x04 or 0x08.
- Support for transaction IDs must be disabled upon receipt of a `RequestIdentify` command, but enabled again before the accessory sends a subsequent `StartIDPS` command.
- Support for transaction IDs must be disabled before sending an `IdentifyDeviceLingoes` command.

Note: When responding to an accessory's command containing a transaction ID, an Apple device with older firmware that does not support transaction IDs will send an `iPodAck` command with a Bad Parameter status (0x04) and no transaction ID.

2.6.1.3 Apple Device Acknowledgment of Transaction ID Commands

To support commands with transaction IDs, the General lingo `iPodAck` command has been expanded to five formats, listed in Table 2-11 (page 112). The new command details are presented in "Command 0x02: iPodAck" (page 124).

Table 2-11 Forms of the General lingo `iPodAck` command

Transaction ID	Command pending	Payload length	Authentication	Format
No	No	0x04	None	Table 3-4 (page 124)
Yes	No	0x06	Version 2.0B	Table 3-5 (page 125)
No	Yes	0x08	None	Table 3-7 (page 126)
Yes	Yes	0x0A	Version 2.0B	Table 3-8 (page 126)
Yes	No	0x0C	Version 2.0B	Table 3-9 (page 127)

2.6.1.4 Applicability of Transaction IDs

When transaction IDs are enabled, all iAP commands must use them. If the description of the command in this specification does not include transaction ID fields, the developer must add them after the Command ID field in every command packet, and every packet length byte must be increased by 2. There are three exceptions to this requirement; the `RequestIdentify` (General lingo command 0x00) command, sent by the Apple device, and the following two accessory responses must omit transaction IDs even though they may be enabled for all other commands:

`Identify` (General lingo command 0x01)
`IdentifyDeviceLingoes` (General lingo command 0x13)

After sending one of these commands the accessory must disable transaction ID support, as described in "Enabling and Disabling Transaction ID Support" (page 111), until it sends a `StartIDPS` command.

2.6.2 Examples of Transaction IDs

Table 2-12 (page 113) through Table 2-15 (page 115) illustrate the use of transaction IDs in some typical iAP command flows.

Table 2-12 Example of IDPS and authentication

Step	Accessory command	Apple device command	Comment
1	StartIDPS		Transaction ID = 0x0001 (accessory counter starts counting).
2		iPodAck	Transaction ID = 0x0001; response to Step 1
3	RequestTransport-MaxPayloadSize		Transaction ID = 0x0002.
4		ReturnTransport-MaxPayloadSize	Transaction ID = 0x0002.
5	SetFIDTokenValues		Transaction ID = 0x0003.
6		AckFIDTokenValues	Transaction ID = 0x0003; assume some token value was not acknowledged.
7	SetFIDTokenValues		Transaction ID = 0x0004; retry command with a different transaction ID
8		AckFIDTokenValues	Transaction ID = 0x0004; response to Step 7.
9	EndIDPS		Transaction ID = 0x0005.
10		IDPSStatus	Transaction ID = 0x0005; response to Step 9
11		GetAccessory-AuthenticationInfo	Transaction ID = 0x0001; first command initiated by the Apple device (Apple device counter starts counting).
12	RetAccessory-AuthenticationInfo		Transaction ID = 0x0001; response to Step 11.
13		AckAccessory-AuthenticationInfo	Transaction ID = 0x0001; response to Step 12.

Table 2-13 Example of entering Extended Interface mode

Step	Accessory command	Apple device command	Comment
1	StartIDPS		Transaction ID = 0x0001 (accessory counter starts counting).
2		iPodAck	Transaction ID = 0x0001; response to Step 1
3	RequestTransport-MaxPayloadSize		Transaction ID = 0x0002.
4		ReturnTransport-MaxPayloadSize	Transaction ID = 0x0002.

Step	Accessory command	Apple device command	Comment
5	SetFIDTokenValues		Transaction ID = 0x0003.
6		AckFIDTokenValues	Transaction ID = 0x0003; response to Step 5.
7	EndIDPS		Transaction ID = 0x0004.
8		IDPSStatus	Transaction ID = 0x0004; response to Step 7.
Enter background authentication state; see "Authentication" (page 103).			
9	SetUIMode		Transaction ID = 0x0005.
10		iPodAck	Transaction ID = 0x0005; response to Step 9: command pending.
11		iPodAck	Transaction ID = 0x0005; sent after the Apple device has entered Extended Interface mode. Note that the same transaction ID value is used.

Table 2-14 Example of getting track artwork data

Step	Accessory command	Apple device command	Comment
1	StartIDPS		Transaction ID = 0x0001 (accessory counter starts counting).
2		iPodAck	Transaction ID = 0x0001; response to Step 1
3	RequestTransport-MaxPayloadSize		Transaction ID = 0x0002.
4		ReturnTransport-MaxPayloadSize	Transaction ID = 0x0002.
5	SetFIDTokenValues		Transaction ID = 0x0003.
6		AckFIDTokenValues	Transaction ID = 0x0003; response to Step 5.
7	EndIDPS		Transaction ID = 0x0004.
8		IDPSStatus	Transaction ID = 0x0004; response to Step 7.
Enter background authentication state; see "Authentication" (page 103).			
9	GetTrackArtworkData		Transaction ID = 0x0005.

Step	Accessory command	Apple device command	Comment
10		RetTrackArtworkData	Transaction ID = 0x0005; Response is divided into multiple packets with the same transaction ID.
11		RetTrackArtworkData	Transaction ID = 0x0005.
12		RetTrackArtworkData	Transaction ID = 0x0005; Last packet returning artwork data.

Table 2-15 Example of sending notifications

Step	Accessory command	Apple device command	Comment
1	StartIDPS		Transaction ID = 0x0001 (accessory counter starts counting).
2		iPodAck	Transaction ID = 0x0001; response to Step 1
3	RequestTransport-MaxPayloadSize		Transaction ID = 0x0002.
4		ReturnTransport-MaxPayloadSize	Transaction ID = 0x0002.
5	SetFIDTokenValues		Transaction ID = 0x0003.
6		AckFIDTokenValues	Transaction ID = 0x0003; response to Step 5.
7	EndIDPS		Transaction ID = 0x0004.
8		IDPSStatus	Transaction ID = 0x0004; response to Step 7.
9		iPodNotification	Transaction ID = 0x0001; first command initiated by the Apple device (Apple device counter starts counting).
10		iPodNotification	Transaction ID = 0x0002; second command initiated by the Apple device.
11	RdsReadyNotify		Transaction ID = 0x0005; next accessory transaction ID.
12	RdsReadyNotify		Transaction ID = 0x0006.

2.7 Multisection Data Transfers

Some iAP commands may need to transfer amounts of data larger than the data receiver's maximum incoming packet size. This section specifies how large amounts of data are transferred in sections, using multiple packets of the same command.

2.7.1 Multisection Transfer Process

The packets of commands that can transfer data in multiple sections contain two 16-bit parameters that uniquely identify transferred data sections:

- Parameter `sectCur` is the index number of the data section in that packet. The first packet has index 0x0000.
- Parameter `sectMax` is the index number of the last data section

When a data transfer fits into a single packet, both `sectCur` and `sectMax` must be set to 0x0000. When a command transfers data in multiple sections, the transaction ID of the command that requested the transfer must be used for all response sections and the `sectCur` value of each serial packet must increment by 0x0001. For optimal data throughput, data sections must be the maximum size permitted by the receiver, less any command header overhead. The last data section may be smaller than the maximum size if the total data transfer size is not a multiple of the section size.

An accessory may either send or receive multisection data. When the accessory sends multisection data to an Apple device, using `RetAccessoryData` or `AsyncAccessoryData`, it must wait after each packet for the Apple device to reply with an `iPodAck` command, as detailed in "Multisection iPodAck Command" (page 389). When the Apple device sends multisection data to an accessory, using `SetAccessoryData`, the accessory must reply with an `AccessoryAck` command after each packet, as detailed in "Multisection AccessoryAck Packet" (page 375).

Regardless of whether the accessory is sender or receiver, the acknowledgment command must have the same transaction ID as the data transfer command. When acknowledging any packet before the last one, the acknowledgment must also return the packet's `sectCur` value plus an `ackStatus` of Section Received OK (0x13) if the section was transferred successfully. When the last section has been received successfully, the receiver must send an `ackStatus` of Command OK (0x00), without a `sectCur` value, to indicate that the entire data transfer has finished.

If an error occurs during a multisection transfer, the receiver must send an acknowledgment with a nonzero `ackStatus` other than 0x13. This means the transfer has failed. The sender must stop sending data sections with the same transaction ID. If the receiver does not acknowledge a multisection packet within 400 ms, the transfer has also failed; the sender must send the receiver an acknowledgment with the transfer's transaction ID and a Command Timeout (0x0F) status. When a transfer fails, the sender can try to send the same data again, starting from the beginning, with a different transaction ID.

2.7.2 Interleaved Transfers

An accessory that makes use of multisection transfers must be capable of handling such transfers even when interleaved with other iAP traffic. (For example, Apple device notifications will be sent while an album artwork transfer is in progress.) The accessory must signal this capability to the Apple device during the identification process (see Table 3-71 (page 164)).

2.8 Apple Device Language

An accessory can use the General lingo commands 0x67 (`GetLocalizationInfo`) and 0x68 (`RetLocalizationInfo`) to determine the language and regional localization of its attached Apple device. If possible, it should then set its text handling and user interface elements to a compatible localization.

2.9 Character Encoding

The iTunes application and all Apple devices use Unicode UTF-8 encoding for tags and metadata. UTF-8 is compatible with the ASCII character set and supports other languages, including Chinese, Japanese, and Korean. It specifies unique encodings to represent all its supported character glyphs including the Roman alphabet and Han, Kanji, Kana, and Hangul characters. The UTF-8 format uses 1 to 4 bytes to represent each encoding; this means that a 5-character word can require from 5 to 20 bytes of storage. More information on Unicode and UTF-8 encoding can be found at http://www.unicode.org/.

Every accessory should decode UTF-8 characters. If the accessory is not capable of displaying certain Unicode character glyphs, a substitute glyph (such as an open square) should be displayed instead.

IMPORTANT: Accessories must make a special provision to decode and render the Unicode RIGHT SINGLE QUOTATION MARK character, which is used extensively by iTunes. This character appears as the 3-byte UTF-8 sequence 0xE2 0x80 0x99. Accessories must recognize and render this sequence using the RIGHT SINGLE QUOTATION MARK character (Unicode 0x2019) or convert it into the simple apostrophe character APOSTROPHE (Unicode 0x0027).

3. The General Lingo

This chapter gives detailed descriptions of the commands included in the General lingo, Lingo 0x00. The General lingo is intended for housekeeping commands and must be supported by all accessories.

3.1 General Lingo Command Summary

Table 3-1 (page 119) gives a summary of all commands in the General lingo, including the command ID, the length of the associated data, and the first version of the General lingo protocol in which the command was supported.

Table 3-1 General lingo commands

Command	ID	Data length without transaction ID	First protocol version
`RequestIdentify`	0x00	0x00	All
`Identify` (**deprecated;** see "General Lingo Command 0x01: Identify" (page 530))	0x01	0x01	All
`iPodAck`	0x02	0x02 or 0x06	1.00
`RequestExtendedInterfaceMode`	0x03	0x00	1.00
`ReturnExtendedInterfaceMode`	0x04	0x01	1.00
`EnterExtendedInterfaceMode` (**deprecated;** use `SetUIMode`)	0x05		
`ExitExtendedInterfaceMode`	0x06	0x00	1.00
`RequestiPodName`	0x07	0x00	1.00
`ReturniPodName`	0x08	0x*NN*	1.00
`RequestiPodSoftwareVersion`	0x09	0x00	1.00
`ReturniPodSoftwareVersion`	0x0A	0x03	1.00
`RequestiPodSerialNum`	0x0B	0x00	1.00
`ReturniPodSerialNum`	0x0C	0x*NN*	1.00
`RequestiPodModelNum` (**deprecated;** use `GetiPodOptionsForLingo`)	0x0D		

Command	ID	Data length without transaction ID	First protocol version
`ReturniPodModelNum` (**deprecated;** use `RetiPodOptionsForLingo`)	0x0E		
`RequestLingoProtocolVersion`	0x0F	0x01	1.00
`ReturnLingoProtocolVersion`	0x10	0x03	1.00
`RequestTransportMaxPayloadSize`	0x11	0x00	1.09
`ReturnTransportMaxPayloadSize`	0x12	0x02	1.09
`IdentifyDeviceLingoes`	0x13	0x0C	1.01
`GetAccessoryAuthenticationInfo`	0x14	0x00	1.01
`RetAccessoryAuthenticationInfo`	0x15	0x*NN*	1.01
`AckAccessoryAuthenticationInfo`	0x16	0x01	1.01
`GetAccessoryAuthenticationSignature`	0x17	0x11 or 0x15	1.01
`RetAccessoryAuthenticationSignature`	0x18	0x*NN*	1.01
`AckAccessoryAuthenticationStatus`	0x19	0x01	1.01
`GetiPodAuthenticationInfo`	0x1A	0x00	1.01
`RetiPodAuthenticationInfo`	0x1B	0x*NN*	1.01
`AckiPodAuthenticationInfo`	0x1C	0x01	1.01
`GetiPodAuthenticationSignature`	0x1D	0x*NN*	1.01
`RetiPodAuthenticationSignature`	0x1E	0x*NN*	1.01
`AckiPodAuthenticationStatus`	0x1F	0x01	1.01
Reserved	0x20–0x22	N/A	N/A
`NotifyiPodStateChange`	0x23	0x01	1.02
`GetiPodOptions`	0x24	0x00	1.05
`RetiPodOptions`	0x25	0x08	1.05
Reserved	0x26	N/A	N/A
`GetAccessoryInfo`	0x27	0x*NN*	1.04
`RetAccessoryInfo`	0x28	0x*NN*	1.04
`GetiPodPreferences`	0x29	0x01	1.05

Command	ID	Data length without transaction ID	First protocol version
RetiPodPreferences	0x2A	0x02	1.05
SetiPodPreferences	0x2B	0x03	1.05
Reserved	0x2C–0x34	N/A	N/A
GetUIMode	0x35	0x00	1.09
RetUIMode	0x36	0x01	1.09
SetUIMode	0x37	0x01	1.09
StartIDPS	0x38	0x00	1.09
SetFIDTokenValues	0x39	0x*NN*	1.09
AckFIDTokenValues	0x3A	0x*NN*	1.09
EndIDPS	0x3B	0x01	1.09
IDPSStatus	0x3C	0x01	1.09
Reserved	0x3D–0x3E	N/A	N/A
OpenDataSessionForProtocol	0x3F	0x05	1.09
CloseDataSession	0x40	0x04	1.09
AccessoryAck	0x41	0x04	1.09
AccessoryDataTransfer	0x42	0x*NN*	1.09
iPodDataTransfer	0x43	0x*NN*	1.09
Reserved	0x44–0x45	N/A	N/A
SetAccessoryStatusNotification	0x46	0x04	1.09
RetAccessoryStatusNotification	0x47	0x04	1.09
AccessoryStatusNotification	0x48	0x*NN*	1.09
SetEventNotification	0x49	0x0A	1.09
iPodNotification	0x4A	0x*NN*	1.09
GetiPodOptionsForLingo	0x4B	0x01	1.09
RetiPodOptionsForLingo	0x4C	0x09	1.09
GetEventNotification	0x4D	0x00	1.09
RetEventNotification	0x4E	0x08	1.09

Command	ID	Data length without transaction ID	First protocol version
GetSupportedEventNotification	0x4F	0x00	1.09
CancelCommand	0x50	0x05	1.09
RetSupportedEventNotification	0x51	0x08	1.09
Reserved	0x52–0x53	N/A	N/A
SetAvailableCurrent	0x54	0x02	1.09
Reserved	0x55	N/A	N/A
SetInternalBatteryChargingState	0x56	0x01	1.09
Reserved	0x57–0x63	N/A	N/A
RequestApplicationLaunch	0x64	0x*NN*	1.09
GetNowPlayingApplicationBundleName	0x65	0x00	1.09
RetNowPlayingApplicationBundleName	0x66	0x*NN*	1.09
GetLocalizationInfo	0x67	0x01	1.09
RetLocalizationInfo	0x68	0x*NN*	1.09
RequestWiFiConnectionInfo	0x69	0x00	1.09
WiFiConnectionInfo	0x6A	0x*NN*	1.09
Reserved	0x6B–0xFF	N/A	N/A

When first connected to an Apple device, every accessory must identify itself. New accessory designs must use the Identify Device Preferences and Settings (IDPS) process described in "Identification" (page 93). This process ensures their compatibility with future firmware. Existing accessory designs may send the iAP General lingo command 0x13, `IdentifyDeviceLingoes`, identifying the lingoes they support. For details, see "Transport Initialization" (page 88).

The Apple device may send a `RequestIdentify` command to the accessory to ask it to reidentify itself. There is currently no data defined for this command. Any command returned in response to a `RequestIdentify` packet does not need to have the extra sync bytes and delays used during the accessory startup process (described in "Transport Initialization" (page 88)).

The remaining General lingo commands can be used to obtain general information from the Apple device. These commands allow the accessory to request the name, serial number, and software version number of the Apple device. The `RequestLingoProtocolVersion` command allows an accessory to query the Apple device for the lingo protocol versions of all supported lingoes on the Apple device. The `iPodAck` command is used by the Apple device to report command error conditions and has a iPodAck Pending feature to notify the requesting accessory how long to wait for responses to certain commands.

Accessories should send the `GetiPodOptionsForLingo` command to query the Apple device for its support of specific features for each lingo, as described in "Determining Apple Device Capabilities" (page 97). The Apple device normally responds with a `RetiPodOptionsForLingo` command, telling the accessory exactly which options it supports for the specific lingo. With Apple devices that do not support `GetiPodOptionsForLingo`, an accessory can send `RequestLingoProtocolVersion` to get the lingo version; however, this method forces the accessory to infer features using the method described in "Protocol Features and Availability" (page 41).

3.2 History of the General lingo protocol

Table 3-2 (page 123) lists changes introduced with each version of the General lingo protocol.

Table 3-2 General lingo revision history

Lingo version	Command changes	Features
No version	Add: 0x00, 0x01	Request identification, identify as single lingo accessory
1.00	Add: 0x02, 0x07–0x10	iPodAck response, request Apple device name, software version, serial number, and model number; support for 38400 and 57600 baud rates
1.01	Add: 0x13–0x1F	Identify as multilingo accessory, Authentication 1.0 process
1.02	Add: 0x23	Notify accessory of Apple device state changes (Sleep/Power On/Hibernate)
1.03	None	(Internal code restructuring)
1.04	Add: 0x27, 0x28	Get/return accessory information, Authentication 2.0 process
1.05	Add: 0x24–0x25, 0x29–0x2B	Video browsing preferences
1.06	None	Line-out preferences, two-way Authentication 2.0
1.07	None	Additional video preferences
1.08	None	Different timeout times for `GetAccessory-AuthenticationInfo` and `GetAccessory-AuthenticationSignature`.
1.09 (see Note, below)	Add: 0x11-0x12, 0x35-0x3C, 0x3F-0x43, 0x4A-0x4C, 0x50, 0x54, 0x64-0x6A	Report maximum payload sizes, identify accessories through IDPS, communicate with applications, query Apple devices for lingo options, support USB Host mode, support iPod Out, support accessory autolaunch, query Apple devices for localization information, support WiFi Network Login Sharing, manage battery usage.

Note: The accessory must use `GetiPodOptionsForLingo` to determine whether the attached Apple device supports specific features of lingo version 1.09.

3.3 General Lingo Commands

This section describes the General lingo commands in detail.

3.3.1 Command 0x00: RequestIdentify

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command to prompt accessories to reidentify themselves. If an accessory receives this command, it must respond with `StartIDPS`, as described in "Transport Initialization" (page 88).

Table 3-3 `RequestIdentify` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

3.3.2 Command 0x02: iPodAck

Lingo: 0x00 — Origin: Apple device

The Apple device sends the `iPodAck` command to notify the accessory of command completion status and errors. The `iPodAck` command may come in one of several forms, depending on the status being returned. For any status other than command pending or dropped data, the iPodAck packet shown in Table 3-4 (page 124) is used. If transaction IDs are enabled, the iPodAck packet shown in Table 3-5 (page 125) is used. Other formats are used for command pending and dropped data, as shown in Table 3-6 (page 125). Transaction IDs are described in "Using Transaction IDs" (page 110).

Table 3-4 `iPodAck` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command status; see Table 3-6 (page 125).
0x*NN*	1	The ID of the command being acknowledged
Packet footer specified in "Command Packets" (page 109).		

Table 3-5 `iPodAck` packet with transaction IDs

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command status; see Table 3-6 (page 125).
0x*NN*	1	`origCmdID`: ID of command received.
Packet footer specified in "Command Packets" (page 109).		

Table 3-6 iAP command error codes

Value	Description
0x00	Success (OK)
0x01	ERROR: Unknown database category or session ID
0x02	ERROR: Command failed
0x03	ERROR: Apple device out of resources
0x04	ERROR: Bad parameter
0x05	ERROR: Unknown ID
0x06	Command Pending; see Table 3-7 (page 126) and Table 3-8 (page 126)
0x07	ERROR: Not authenticated
0x08	ERROR: Bad authentication version
0x09	ERROR: Accessory power mode request failed
0x0A	ERROR: Certificate invalid
0x0B	ERROR: Certificate permissions invalid
0x0C	ERROR: File is in use
0x0D	ERROR: Invalid file handle
0x0E	ERROR: Directory not empty
0x0F	ERROR: Operation timed out
0x10	ERROR: Command unavailable in this Apple device mode
0x11	ERROR: Accessory Detect not grounded, or invalid Accessory Identify Resistor detected
0x12	ERROR: Selection not Genius
0x13	Multisection data section received successfully; see "Multisection Data Transfers" (page 116)

Value	Description
0x14	ERROR: Lingo busy
0x15	ERROR: Maximum number of accessory connections already reached
0x16	HID descriptor index already in use
0x17	ERROR: Dropped data (`SessionWriteFailure`); see Table 3-9 (page 127)
0x18	ERROR: Attempt to enter iPod Out mode with incompatible video settings
0x19–0xFF	Reserved

If the status returned by the `iPodAck` command is Command Pending, an additional field is added to the `iPodAck` packet that represents the amount of time, in milliseconds, that an accessory should wait to receive the final packet indicating that the current command completed or returned an error status.

After receiving a Command Pending `iPodAck`, the accessory should wait for up to the specified number of milliseconds for a final `iPodAck` response. If no final `iPodAck` packet is received before the specified amount of time expires, the accessory should retry the command.

Table 3-7 `iPodAck` packet with Command Pending status but no transaction ID

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x06	1	Command result status: Command Pending
0x*NN*	1	The ID of the command being acknowledged.
0x*NN*	4	Maximum amount of time to wait for pending response, in milliseconds .
Packet footer specified in "Command Packets" (page 109).		

Table 3-8 `iPodAck` packet with Command Pending status and transaction ID

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x06	1	Command result status: Command Pending
0x*NN*	1	The ID of the command being acknowledged.
0x*NN*	4	Maximum amount of time to wait for pending response, in milliseconds .
Packet footer specified in "Command Packets" (page 109).		

If the `iPodAck` command returns `SessionWriteFailure` (error code 0x17), the Apple device was unable to send all the data required by an `AccessoryDataTransfer` command. The Apple device adds fields to the `iPodAck` packet, as shown in Table 3-9 (page 127), to indicate which `AccessoryDataTransfer` session ID failed and how many bytes were dropped from the end of the packet that the accessory sent. The accessory may assume that the Apple device successfully accepted the earlier bytes.

Note: The `SessionWriteFailure` return is supported only in products running iOS 4 and later versions.

Table 3-9 `iPodAck` packet reporting dropped data

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x17	1	Command result: `SessionWriteFailure`
0x42	1	The ID of the command being acknowledged (`AccessoryDataTransfer`).
0x*NN*	2	`sessionID`: Session ID used by `AccessoryDataTransfer`
0x*NN*	4	`numBytesDropped`
Packet footer specified in "Command Packets" (page 109).		

3.3.3 Command 0x03: RequestExtendedInterfaceMode

Lingo: 0x00 — Origin: Accessory

The accessory requests Extended Interface mode from the Apple device. The Apple device responds with "Command 0x04: ReturnExtendedInterfaceMode" (page 127). This command may be used only if the accessory requests Lingo 0x04 during its identification process.

Table 3-10 `RequestExtendedInterfaceMode` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

3.3.4 Command 0x04: ReturnExtendedInterfaceMode

Lingo: 0x00 — Origin: Apple device

The Apple device reports whether its current operating mode is Extended Interface mode. If the returned mode byte is nonzero (true), the Apple device is in Extended Interface mode; otherwise it is in another mode (standard mode or iPod Out mode). This command may be used only if the accessory requests Lingo 0x04 during its identification process.

Table 3-11 `ReturnExtendedInterfaceMode` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Mode byte. If nonzero (true), the Apple device is in Extended Interface mode; if zero (false), it is not.
Packet footer specified in "Command Packets" (page 109).		

3.3.5 Command 0x06: ExitExtendedInterfaceMode

Lingo: 0x00 — Origin: Accessory

The accessory sends this command to the Apple device to force it to exit the Extended Interface mode. If the Apple device is already in the Standard UI mode, it immediately returns a General lingo `iPodAck` command packet, notifying the user that the command was successful. This command may be used only if the accessory requests Lingo 0x04 during its identification process.

If the Apple device needs to switch modes, it sends a General lingo iPodAck Pending command packet informing the accessory how long it will take the Apple device to switch modes. This is followed by an `iPodAck` packet notifying the accessory that the Apple device successfully changed modes. Accessories should honor the timeout returned by the iPodAck Pending before assuming the original command has failed.

Table 3-12 `ExitExtendedInterfaceMode` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

3.3.6 Command 0x07: RequestiPodName

Lingo: 0x00 — Origin: Accessory

This command retrieves the name of the Apple device. The Apple device responds with a "Command 0x08: ReturniPodName" (page 129) command containing the name of the Apple device as a null-terminated UTF-8 character array.

Table 3-13 `RequestiPodName` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		

Value	Bytes	Parameter
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

3.3.7 Command 0x08: ReturniPodName

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command in response to the "Command 0x07: RequestiPodName" (page 128) message from the accessory. The Apple device name is encoded as a null-terminated UTF-8 character array. If the Apple device name has not been modified by the user, it is returned as "iPod".

Note: Starting with version 1.02 of the General lingo, the `ReturniPodName` command on Windows-formatted Apple devices returns the iTunes name of the Apple device instead of the Windows volume name.

Table 3-14 `ReturniPodName` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	*NN*	The name of the Apple device as a null-terminated UTF-8 character array.
Packet footer specified in "Command Packets" (page 109).		

3.3.8 Command 0x09: RequestiPodSoftwareVersion

Lingo: 0x00 — Origin: Accessory

This command retrieves the software version information for the Apple device. The Apple device responds with a "Command 0x0A: ReturniPodSoftwareVersion" (page 130) containing the major, minor, and revision version numbers.

Note: This command requests the Apple device software version and not the version of a particular lingo protocol.

Table 3-15 `RequestiPodSoftwareVersion` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

3.3.9 Command 0x0A: ReturniPodSoftwareVersion

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command in response to the "Command 0x09: RequestiPodSoftwareVersion" (page 129) message from the accessory. The Apple device returns each version number as an individual byte, with the major version number sent first. For example, if the major, minor, and revision bytes are returned as 0x01, 0x02, and 0x03, the Apple device software version number is 1.02.03.

Table 3-16 `ReturniPodSoftwareVersion` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Apple device major version number.
0x*NN*	1	Apple device minor version number.
0x*NN*	1	Apple device revision version number.
Packet footer specified in "Command Packets" (page 109).		

3.3.10 Command 0x0B: RequestiPodSerialNum

Lingo: 0x00 — Origin: Accessory

This command retrieves the serial number string of the Apple device. The Apple device responds with a "Command 0x0C: ReturniPodSerialNum" (page 130) containing the serial number as a null-terminated UTF-8 character array.

Table 3-17 `RequestiPodSerialNum` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

3.3.11 Command 0x0C: ReturniPodSerialNum

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command in response to the "Command 0x0B: RequestiPodSerialNum" (page 130) message from the accessory. The Apple device serial number is encoded as a null-terminated UTF-8 character array.

Table 3-18 `ReturniPodSerialNum` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*…	*NN*	The Apple device serial number, as a null-terminated UTF-8 character array.
Packet footer specified in "Command Packets" (page 109).		

3.3.12 Command 0x0F: RequestLingoProtocolVersion

Lingo: 0x00 — Origin: Accessory

This command retrieves version information for any of the lingoes supported by the Apple device. The Apple device responds with a "Command 0x10: ReturnLingoProtocolVersion" (page 131) containing the major and minor version information of the requested Apple device lingo. This command has one parameter, the lingo whose version information is requested. The Apple device returns an `iPodAck` command with a bad parameter status if an accessory calls this command with an invalid or unsupported lingo ID. When an Apple device does not respond to the `GetiPodOptionsForLingo` command, the accessory may use the `RequestLingoProtocolVersion` command to determine what iAP features are available for each lingo used.

Table 3-19 `RequestLingoProtocolVersion` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	The lingo for which to request version information.
Packet footer specified in "Command Packets" (page 109).		

3.3.13 Command 0x10: ReturnLingoProtocolVersion

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command in response to the "Command 0x0F: RequestLingoProtocolVersion" (page 131) message from the accessory. The major and minor version information for the requested lingo are returned.

Table 3-20 `ReturnLingoProtocolVersion` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	The lingo for which version information is being returned.
0x*NN*	1	The major protocol version for the given lingo.

Value	Bytes	Parameter
0x*NN*	1	The minor protocol version for the given lingo.
Packet footer specified in "Command Packets" (page 109).		

3.3.14 Command 0x11: RequestTransportMaxPayloadSize

Lingo: 0x00 — Origin: Accessory

An accessory sends this command to an Apple device to determine the maximum allowable payload size per packet using the current iAP transport (UART, USB Device mode, USB Host mode, or Bluetooth). The Apple device replies by sending a `ReturnTransportMaxPayloadSize` command. If the Apple device returns an `iPodAck` command with an error value of 0x04 (Bad Parameter), then the maximum payload size is the default value of 506 bytes. A packet's payload size is equal to the value of its `length` field, as defined in "Payload Length Field" (page 110).

Table 3-21 `RequestTransportMaxPayloadSize` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

3.3.15 Command 0x12: ReturnTransportMaxPayloadSize

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command in response to a `RequestTransportMaxPayloadSize` command, to tell the accessory the maximum allowable packet payload size, in bytes, for the current transport.

Table 3-22 `ReturnTransportMaxPayloadSize` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	2	`maxPayload`: The maximum allowable packet payload, in bytes. A packet's payload size is equal to the value of its `length` field, as defined in "Payload Length Field" (page 110).
Packet footer specified in "Command Packets" (page 109).		

3.3.16 Command 0x13: IdentifyDeviceLingoes

Lingo: 0x00 — Origin: Accessory

IMPORTANT: New accessory designs must use the Identify Device Preferences and Settings (IDPS) process described in "Identification" (page 93). This process ensures their compatibility with future firmware. Existing accessory designs may continue to send the `IdentifyDeviceLingoes` command, identifying the lingoes they support and requesting immediate authentication.

The accessory sends this command to signal its presence and to identify its supported lingoes. In response, the Apple device sends an `iPodAck` command. Accessories use the `IdentifyDeviceLingoes` command to report all supported lingoes; it must be used in place of the `Identify` (0x01) command.

Note: The `IdentifyDeviceLingoes` command may also be used to make the Apple device display an "Accessory not supported" message; see the response to Step 6 in Table 2-4 (page 98).

The `IdentifyDeviceLingoes` command resets all accessory information set by a previous `Identify` command, including the authentication retry counter and any previously granted authentication access permissions. The payload of this command includes three fields: the Device Lingoes Spoken, Options, and Device ID fields.

Note: An accessory attached via the 30-pin connector must always monitor the Accessory Power output from the Apple device (pin 13 in "Hardware Interfaces" in *MFi Accessory Hardware Specification*). If Accessory Power goes low, even momentarily, the accessory must stop sending iAP commands. After Accessory Power goes high, the accessory must wait at least 80 ms and then send a `StartIDPS` command, following the steps shown in Table 2-4 (page 98).

The `IdentifyDeviceLingoes` command disables all but free lingoes on the current port unless authentication is requested (immediate authentication is also required for Authentication 2.0). For serial ports, this means lingoes 0x00, 0x02, 0x03, and 0x04 may be used, excluding authenticated commands; the USB port will be able to use only the general lingo, 0x00 (see Table 2-7 (page 104)). Accessories that register with this command can use only those lingoes that they specifically identify (see Table 3-24 (page 134)).

If any lingo identified by the `IdentifyDeviceLingoes` command can be used by only one accessory at a time, and that lingo is already in use by a different accessory, the command will fail but no command failure `iPodAck` command will be sent to the accessory. The accessory can verify that the `IdentifyDeviceLingoes` command has succeeded by sending a free command with valid parameters and checking the Apple device's response. The Apple device will respond with an `iPodAck` response with failure status if any lingo is already in use. An identification failure results in the accessory being able to use only the General lingo (0x00).

This command performs lingo conflict checking to ensure that single-instance lingoes, such as Display Remote, are used on only one port at a time. If the `IdentifyDeviceLingoes` command is acknowledged with a status of 0x15 (Maximum number of accessory connections already reached), the accessory must not send any further iAP traffic until it has been disconnected and reconnected to the Apple device.

For sample command sequences in which an accessory identifies its supported lingoes, using `IdentifyDeviceLingoes`, see "Sample Identification Sequences" (page 389).

Table 3-23 `IdentifyDeviceLingoes` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Device Lingoes Spoken; see Table 3-24 (page 134).
0x*NN*	4	Options; see Table 3-25 (page 135).
0x*NN*	4	Device ID. Accessories must send a unique identifier, supplied by the iPod Authentication Coprocessor, if they require authentication. If an accessory does not require authentication, it can send a Device ID of 0x00000000 and set the authentication option bits to 0x0.
Packet footer specified in "Command Packets" (page 109).		

Use the Device Lingoes Spoken field as a bit field to set each bit corresponding to the lingoes supported by the accessory. For example, if an accessory supports both the Simple Remote and RF Tuner lingoes, the value of the bit field is 0x00000085 and the low byte in binary is 10000101.

Table 3-24 Device Lingoes Spoken bits

Bit	Supported lingo
0	General lingo (must be set by all accessories)
1	Reserved; set to 0
2	Simple Remote lingo
3	Display Remote lingo
4	Extended Interface lingo
5	Reserved; set to 0
6	USB Host Mode lingo
7	RF Tuner lingo
8	Accessory Equalizer lingo
9	Sports lingo
10	Digital Audio lingo
11	Reserved; set to 0
12	Storage lingo
31:13	Reserved; set to 0

Note: The iPod Out lingo (Lingo 0x0D) and the Location lingo (Lingo 0x0E) are available only if the accessory has identified itself using IDPS. See "Transport Initialization" (page 88).

The bits of the Options field are defined as shown in Table 3-25 (page 135).

Table 3-25 `IdentifyDeviceLingoes` Options bits

Bits	Meaning
1:0	Authentication control bits. These bits have the following meanings: 00 = no authentication is supported or required 01 = defer authentication until an authenticated command is used (Authentication 1.0 only); see Note below 10 = authenticate immediately after identification (required for Authentication 2.0). 11 = reserved
31:2	Reserved; set to 0

Accessories identifying using the `IdentifyDeviceLingoes` command receive notifications when the Apple device changes state. See "Command 0x23: NotifyiPodStateChange" (page 143) for more details.

Note: If an accessory uses an invalid Device ID during an identification attempt, the Apple device returns a bad parameter error (0x04) `iPodAck`. If the accessory claims a Device Lingo Spoken that is not supported by the attached Apple device, the Apple device returns a command failed (0x02) `iPodAck`.

3.3.16.1 Canceling a Current Authentication Process With IdentifyDeviceLingoes

For best success calling `IdentifyDeviceLingoes`, use the following sequence:

1. For UART-based communications, ensure that the Accessory Identify pin (pin 10) is connected to the correct resistors and that the Accessory Detect (pin 20) pin is grounded; see "Accessory Detect and Identify" in *MFi Accessory Hardware Specification*. Also, precede all iAP packets by an extra 0xFF autobaud synchronization byte.
2. The accessory sends an "artificial" `IdentifyDeviceLingoes` command with the lingo mask set to only the General lingo, the option bitmask set to 0x0 (no options), and the Device ID set to 0x0. This will cancel any active authentication process on the current Apple device accessory port.
3. The accessory waits up to 500 ms for the Apple device to send an `iPodAck` success response. If this does not happen within 500 ms, the accessory may return to Step 2 as many times as desired.
4. The Apple device follows its `iPodAck` success response by sending a `GetAccessoryInfo` command with a valid Accessory Info Type parameter.
5. The accessory must respond with a `RetAccessoryInfo` command containing the requested information. This information will be superseded later when the accessory sends `RetAccessoryInfo` again during authentication. See Table 2-8 (page 106).

6. The accessory now sends a "genuine" `IdentifyDeviceLingoes` command, passing the correct parameters, and proceeds with identification and authentication.

Sample command sequences that illustrate some of these best practices are listed in Table C-48 (page 553) and Table C-49 (page 555).

3.3.17 Command 0x14: GetAccessoryAuthenticationInfo

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command to obtain authentication information from the accessory. The command is sent only if the accessory has indicated that it supports authentication during its identification process and has passed a valid, nonzero Device ID. In response, the accessory sends "Command 0x15: RetAccessoryAuthenticationInfo" (page 136).

Table 3-26 `GetAccessoryAuthenticationInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

3.3.18 Command 0x15: RetAccessoryAuthenticationInfo

Lingo: 0x00 — Origin: Accessory

The accessory indicates the iAP authentication version that it supports by returning this command in response to a "Command 0x14: GetAccessoryAuthenticationInfo" (page 136) command from the Apple device. The authentication version returned by this command must be consistent with the Device ID sent during its identification process.

The returned packet may have either of two formats, depending on whether the authentication process is Authentication 1.0 or 2.0. See Table 3-27 (page 136) and Table 3-28 (page 137).

Table 3-27 `RetAccessoryAuthenticationInfo` packet, Authentication 1.0

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Authentication protocol major version number
0x*NN*	1	Authentication protocol minor version number
Packet footer specified in "Command Packets" (page 109).		

Table 3-28 `RetAccessoryAuthenticationInfo` packet, Authentication 2.0

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x02	1	Authentication major version (0x02)
0x00	1	Authentication minor version (0x00)
0x*NN*	1	X.509 certificate current section index (0 = first section, 1 = second section, and so on)
0x*NN*	1	X.509 certificate maximum section index (0 = one section total, 1 = two sections, and so on)
0x*NN*	*NN*	X.509 certificate data. If the data length exceeds the value determined by calling `RequestTransportMaxPayloadSize`, the data must be broken into sections.
Packet footer specified in "Command Packets" (page 109).		

During Authentication 2.0, the number of certificate bytes sent at once may not exceed the lesser of the value determined by calling `RequestTransportMaxPayloadSize`, or the maximum payload size allowed by the Apple device's iOS version (2048 for iOS 6.0, 500 for earlier versions). If the X.509 certificate is larger than that amount, the accessory must divide it into sections for reassembly by the Apple device. The Apple device will send an `iPodAck` command for each certificate section up to, but not including, the last one. The accessory must wait for the `iPodAck` command before sending the next certificate section. The final certificate section is acknowledged by an `AckAccessoryAuthenticationInfo` command.

The `iPodAck` response to each certificate section only acknowledges receipt. When all the sections have been assembled, the Apple device evaluates the whole certificate and sends the `AckAccessoryAuthenticationInfo` command that states whether or not it is valid.

3.3.19 Command 0x16: AckAccessoryAuthenticationInfo

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command in response to "Command 0x15: RetAccessoryAuthenticationInfo" (page 136). It indicates the current state of the accessory authentication information. If the accessory receives a nonzero status, the Apple device does not support the accessory's authentication version and the accessory will fail authentication.

Note: The accessory must send certificate data in sections not larger than the maximum determined by calling `RequestTransportMaxPayloadSize` and must wait for an `iPodAck` command after each one. The Apple device acknowledges the final certificate section with this command.

Table 3-29 `AckAccessoryAuthenticationInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		

Value	Bytes	Parameter
0x*NN*	1	Status of authentication information. Possible values are: 0x00 = Authentication information supported 0x04 = Certificate too long or sections out of order 0x08 = Authentication information unsupported 0x0A = Certificate is invalid 0x0B = Certificate permissions are invalid
Packet footer specified in "Command Packets" (page 109).		

3.3.20 Command 0x17: GetAccessoryAuthenticationSignature

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command to authenticate an accessory that has identified itself as requiring authentication. Authentication occurs either immediately upon identification or when the accessory attempts to use a restricted lingo or command. The accessory calculates its digital signature based on the challenge offered by the Apple device and sends the results back to the Apple device using "Command 0x18: RetAccessoryAuthenticationSignature" (page 139).

The authentication retry counter is used to track the number of retries. If the returned signature cannot be verified, the Apple device responds with a nonzero "Command 0x19: AckAccessoryAuthenticationStatus" (page 139), followed immediately by another "Command 0x17: GetAccessoryAuthenticationSignature" (page 138).

The retry counter is set to 0x01 for the first authentication attempt and incremented each time the Apple device retries the `GetAccessoryAuthenticationSignature` command. Accessories using Authentication 1.0 are allowed up to four retries and a maximum of 30 seconds (up to 7.5 seconds per retry) for the authentication process. Accessories using Authentication 2.0 are allowed up to two retries and a maximum of 150 seconds (up to 75 seconds per retry) for the authentication process. If an accessory fails to respond to the `GetAccessoryAuthenticationSignature` command, the authentication process will fail and the accessory will not be able to use any of the authenticated commands.

Table 3-30 `GetAccessoryAuthenticationSignature` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	*NN*	An offered challenge sent for the accessory to sign and return (16 bytes for Authentication 1.0, 20 bytes for Authentication 2.0).
0x*NN*	1	Authentication retry counter. The authentication process terminates after the maximum retry count or maximum timeout interval has been reached, whichever comes first. When the authentication process fails, only nonauthenticated lingoes and commands are usable.
Packet footer specified in "Command Packets" (page 109).		

3.3.21 Command 0x18: RetAccessoryAuthenticationSignature

Lingo: 0x00 — Origin: Accessory

The accessory sends this command to the Apple device in response to "Command 0x17: GetAccessoryAuthenticationSignature" (page 138). The Apple device verifies the digital signature, calculated by the accessory based on the offered challenge. If verification passes, the Apple device authenticates the accessory and updates its lingo and command access permissions accordingly. The authentication status is sent to the accessory using "Command 0x19: AckAccessoryAuthenticationStatus" (page 139).

Table 3-31 `RetAccessoryAuthenticationSignature` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	*NN*	The digital signature calculated by the accessory, based on the offered challenge (variable length).
Packet footer specified in "Command Packets" (page 109).		

3.3.22 Command 0x19: AckAccessoryAuthenticationStatus

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command to the accessory in response to the "Command 0x18: RetAccessoryAuthenticationSignature" (page 139) command. It indicates the current accessory authentication state, as detailed in "Apple Device Authentication of the Accessory" (page 106). If the accessory receives a nonzero status, the accessory has failed authentication and will only be able to use unauthenticated lingo commands.

If the accessory receives a zero status, the Apple device has successfully authenticated the accessory. The accessory may then use the requested authenticated lingoes and commands. Optionally, the accessory may begin the process of authenticating the Apple device, by sending "Command 0x1A: GetiPodAuthenticationInfo" (page 140).

Table 3-32 `AckAccessoryAuthenticationStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Status of the authentication operation: 0x00 = Authentication operation passed All other values = Authentication operation failed; see Table 3-6 (page 125)
Packet footer specified in "Command Packets" (page 109).		

3.3.23 Command 0x1A: GetiPodAuthenticationInfo

Lingo: 0x00 — Origin: Accessory

The accessory sends this command to obtain authentication information from the Apple device. The accessory should send this command only if the accessory has indicated that it supports authentication during its identification process and the Apple device has successfully authenticated the accessory. (Accessory authentication is successful when the accessory receives the "Command 0x19: AckAccessoryAuthenticationStatus" (page 139) command with a status of 0x00). In response to `GetiPodAuthenticationInfo` the Apple device sends "Command 0x1B: RetiPodAuthenticationInfo" (page 140).

Table 3-33 `GetiPodAuthenticationInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

3.3.24 Command 0x1B: RetiPodAuthenticationInfo

Lingo: 0x00 — Origin: Apple device

The Apple device returns this command in response to "Command 0x1A: GetiPodAuthenticationInfo" (page 140) from the accessory.

Note: Authentication version 2.00 (major version 0x02, minor version 0x00) is the only version supported by Apple devices that can be authenticated by accessories.

Table 3-34 `RetiPodAuthenticationInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x02	1	Authentication major version (0x02)
0x00	1	Authentication minor version (0x00)
0x*NN*...	1	X.509 certificate current section index (0 = first section, 1 = second section, and so on)
0x*NN*	1	X.509 certificate maximum section index (0 = one section total, 1 = two sections, and so on)
0x*NN*	*NN*	X.509 certificate data. The data may be broken into sections, each not larger than the maximum determined by calling `RequestTransportMaxPayloadSize`.

Value	Bytes	Parameter
Packet footer specified in "Command Packets" (page 109).		

The accessory must wait until the entire certificate has been sent by the Apple device and then within 15 seconds send an `AckiPodAuthenticationInfo` command in response based on its evaluation of the certificate.

3.3.25 Command 0x1C: AckiPodAuthenticationInfo

Lingo: 0x00 — Origin: Accessory

The accessory sends this command to the Apple device in response to "Command 0x1B: RetiPodAuthenticationInfo" (page 140). It indicates the current state of the Apple device authentication information version. If the accessory sends a nonzero status, it indicates that it will not be able to authenticate the Apple device due to an invalid X.509 certificate.

Table 3-35 `AckiPodAuthenticationInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Status of the authentication information: 0x00 = authentication information valid Other values = authentication information not valid
Packet footer specified in "Command Packets" (page 109).		

3.3.26 Command 0x1D: GetiPodAuthenticationSignature

Lingo: 0x00 — Origin: Accessory

The accessory uses this command to send an offered challenge to the Apple device for digital signature. In response, the Apple device returns its signed challenge to the accessory using "Command 0x1E: RetiPodAuthenticationSignature" (page 142). Accessories must implement the authentication retry feature described in "Command 0x17: GetAccessoryAuthenticationSignature" (page 138), allowing the Apple device two retries in 150 seconds. The retry counter must be set to 0x01 in the first `GetiPodAuthenticationSignature` command sent to the Apple device and must be incremented for each subsequent attempt. Authentication must fail after either the retry count or maximum response interval have been exceeded since the first `GetiPodAuthenticationSignature` command was sent.

Table 3-36 `GetiPodAuthenticationSignature` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		

Value	Bytes	Parameter
0x*NN*	20	An offered challenge for the Apple device to sign and return (20 bytes)
0x*NN*	1	Authentication retry counter. A maximum of two retries or 150 seconds, whichever happens first, should occur before the authentication process is terminated.
Packet footer specified in "Command Packets" (page 109).		

3.3.27 Command 0x1E: RetiPodAuthenticationSignature

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command to the accessory in response to "Command 0x1D: GetiPodAuthenticationSignature" (page 141). The accessory verifies the digital signature, calculated by the Apple device based on the offered challenge, and, if verification passes, authenticates the Apple device. The accessory sends the authentication status to the Apple device using "Command 0x1F: AckiPodAuthenticationStatus" (page 142).

Table 3-37 `RetiPodAuthenticationSignature` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	*NN*	The digital signature calculated by the Apple device
Packet footer specified in "Command Packets" (page 109).		

3.3.28 Command 0x1F: AckiPodAuthenticationStatus

Lingo: 0x00 — Origin: Accessory

The accessory sends this command to the Apple device in response to "Command 0x1E: RetiPodAuthenticationSignature" (page 142). It indicates the current Apple device authentication state. The accessory should return a nonzero `iPodAck` for each failed authentication attempt.

Table 3-38 `AckiPodAuthenticationStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Status of authentication information: 0x00 = authentication operation passed Other values = authentication operation failed
Packet footer specified in "Command Packets" (page 109).		

3.3.29 Command 0x23: NotifyiPodStateChange

Lingo: 0x00 — Origin: Apple device

When the Apple device power state is about to change, the Apple device sends this notification command to accessories that identify using IDPS or `IdentifyDeviceLingoes`. If the accessory identifies using the deprecated command `Identify`, this notification is not sent. The state change byte indicates the specific Apple device state transition. If the Apple device is switching from a Power On state to a Sleep state, accessories must immediately reduce their power consumption to low power mode, as defined in the section "Accessory Power Policy" in *MFi Accessory Hardware Specification*. When the Apple device has transitioned to Hibernate state, self-powered accessories are expected to automatically reidentify themselves 80 ms after current is restored on the Accessory Power line of the 30-pin connector.

Table 3-39 `NotifyiPodStateChange` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`StateChg`: The Apple device state change. Possible values are: 0x00 = reserved. 0x01 = current from the Apple device on the Accessory Power line of the 30-pin connector going to Hibernate state (no power) and not preserving menu selections or playback context (see "Power States" in *MFi Accessory Hardware Specification*). 0x02 = current from the Apple device going to Hibernate state (no power) but preserving menu selections and playback context. 0x03 = current from the Apple device going to Sleep state (drawing less than 5 mA current). 0x04 = current from the Apple device going to the Power On state. 0x05–0xFF = reserved.
Packet footer specified in "Command Packets" (page 109).		

Note: Firmware version 1.0.0 of the iPod nano reported its state change incorrectly. The `StateChg` byte is decremented by 1, so that 0x00 represents a value of 0x01, 0x01 represents a value of 0x02, and so forth. All later versions of the nano firmware conform to the table above.

3.3.30 Command 0x24: GetiPodOptions

Lingo: 0x00 — Origin: Accessory

IMPORTANT: The preferred way to obtain information about the capabilities of an attached Apple device is to send "Command 0x4B: GetiPodOptionsForLingo" (page 191). New accessory designs must send `GetiPodOptionsForLingo` first. If and only if an `iPodAck` command with Bad Parameter status (0x04) is returned, then the accessory may send `GetiPodOptions` instead.

The accessory sends this command to ask the Apple device to return a 64-bit field that defines the options that the Apple device supports. In response, the Apple device sends a `RetiPodOptions` command.

Table 3-40 `GetiPodOptions` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

3.3.31 Command 0x25: RetiPodOptions

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command in response to a `GetiPodOptions` command from the accessory.

Table 3-41 `RetiPodOptions` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	8	Option bits: Bit 1: the Apple device supports using `SetiPodPreferences` to control line-out usage Bit 0: the Apple device supports video output All other bits reserved
Packet footer specified in "Command Packets" (page 109).		

3.3.32 Command 0x27: GetAccessoryInfo

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command to accessories that identify themselves using the `IdentifyDeviceLingoes` command to obtain certain information from the accessory. The accessory must respond with a `RetAccessoryInfo` command for the required accessory Info Types. The Apple device will use the information gathered to:

- Display accessory information in the Settings:About box on the Apple device

- Display a message to the user if the Apple device firmware needs to be updated to support the accessory
- Display a message to the user if the Apple device firmware does not support the accessory

The `GetAccessoryInfo` command sends the Accessory Info Type and the Accessory Info Type parameters to the accessory. Table 3-42 (page 145) lists the number of parameter bytes for the corresponding Accessory Info Types. The Apple device requests each of the Accessory Info Types in the order in which they appear in Table 3-42 (page 145). Because the Accessory minimum supported Apple device firmware version request (which sends the Apple device model number as a parameter) is sent before any Accessory minimum supported lingo version requests, the accessory has the option of changing its responses to those appropriate for the Apple device model.

When the `GetAccessoryInfo` command is sent with the accessory minimum supported lingo version info type, the 1-byte lingo number for which the Apple device is requesting the minimum supported version is sent as a parameter. The Apple device will send the `GetAccessoryInfo` command with this Accessory Info Type for every lingo that the accessory indicates it supports.

The Apple device begins sending `GetAccessoryInfo` commands as soon as an accessory identifies itself successfully via the `IdentifyDeviceLingoes` command and either enters the background authentication state or does not request authentication. If the accessory does not respond, the Apple device waits 5 seconds for a response before timing out and retrying. It retries a maximum of three times.

The types of data that `GetAccessoryInfo` can request are listed in Table 3-42 (page 145). The command's packet has two formats, depending on the data type, as shown in the table.

If an accessory supports `GetAccessoryInfo` requests from the Apple device (Info Type 0x0B), it must register its support during the IDPS process by sending the Apple device an `AccessoryInfoToken` with an `accInfoType` of 0x0B; see Table 3-72 (page 165).

Table 3-42 Accessory Info Type values

Value	Meaning	Parameter bytes	Requirement	Packet format
0x00	Accessory info capabilities	0	Required	Table 3-43 (page 146)
0x01	Accessory name	0	Required	Table 3-43 (page 146)
0x02	Accessory minimum supported iPod firmware version (**deprecated**)	7	Optional	Table 3-44 (page 146)
0x03	Accessory minimum supported lingo version	1	Optional	Table 3-45 (page 146)
0x04	Accessory firmware version	0	Required	Table 3-43 (page 146)
0x05	Accessory hardware version	0	Required	Table 3-43 (page 146)
0x06	Accessory manufacturer	0	Required	Table 3-43 (page 146)
0x07	Accessory model number	0	Required	Table 3-43 (page 146)
0x08	Accessory serial number	0	Optional	Table 3-43 (page 146)

Value	Meaning	Parameter bytes	Requirement	Packet format
0x09	Accessory incoming maximum payload size. A packet's payload size is equal to the value of its `length` field, as defined in "Payload Length Field" (page 110).	0	Optional	Table 3-43 (page 146)
0x0A	Reserved			
0x0B	Accessory status types supported; see Table 3-55 (page 151)	0	Optional	Table 3-43 (page 146)
0x0C-0xFF	Reserved			

Table 3-43 `GetAccessoryInfo` packet with Accessory Info Type = 0x00-0x01, 0x04-0x09, or 0x0B

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Accessory Info Type; see Table 3-42 (page 145)
Packet footer specified in "Command Packets" (page 109).		

Table 3-44 `GetAccessoryInfo` packet with Accessory Info Type = 0x02 (deprecated)

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x02	1	Accessory Info Type (see Table 3-42 (page 145))
0x*NN*	4	Apple device Model ID
Packet footer specified in "Command Packets" (page 109).		

Table 3-45 `GetAccessoryInfo` packet with Accessory Info Type = 0x03

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x03	1	Accessory Info Type (see Table 3-42 (page 145))
0x*NN*	1	Lingo ID
Packet footer specified in "Command Packets" (page 109).		

3.3.33 Command 0x28: RetAccessoryInfo

Lingo: 0x00 — Origin: Accessory

If the accessory is attached to a non-IDPS Apple device, it can reply to a second `GetAccessoryInfo` request with an `IdentifyDeviceLingoes` command, as shown in Step 10 of Table C-48 (page 553).

Except for that case, the accessory must reply with this command to every receipt of `GetAccessoryInfo` from the Apple device. It should reply within 500 ms. The data contained in the packet must be responsive to the Accessory Info Type requested by the Apple device.

When the `RetAccessoryInfo` command is returning the accessory info capabilities, a bit-field is returned where every set bit represents a supported Accessory Info Type. See Table 3-42 (page 145) for a description of the Accessory Info Type; see Table 3-46 (page 147) for the meanings of its returned data bytes. Each Accessory Info Type value must be returned in the appropriate `RetAccessoryInfo` packet format, as shown in the table.

Table 3-46 Accessory Info Type values for RetAccessoryInfo

Value	Meaning	Parameter bytes	Packet format
0x00	Accessory info capabilities	4	Table 3-47 (page 148)
0x01	Accessory name	1–64	Table 3-49 (page 149)
0x02	Accessory minimum supported iPod firmware version (**deprecated**)	7	Table 3-50 (page 149)
0x03	Accessory minimum supported lingo version	3	Table 3-51 (page 150)
0x04	Accessory firmware version	3	Table 3-52 (page 150)
0x05	Accessory hardware version	3	Table 3-52 (page 150)
0x06	Accessory manufacturer	1–64	Table 3-49 (page 149)
0x07	Accessory model number	1–64	Table 3-49 (page 149)
0x08	Accessory serial number	1–64	Table 3-49 (page 149)
0x09	Accessory incoming maximum payload size. A packet's payload size is equal to the value of its `length` field, as defined in "Payload Length Field" (page 110).	2	Table 3-53 (page 151)
0x0A	Reserved	N/A	
0x0B	Accessory status types supported; see Table 3-55 (page 151)	4	Table 3-54 (page 151)
0x0C-0xFF	Reserved	N/A	

Table 3-47 `RetAccessoryInfo` packet with Accessory Info Type = 0x00

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x00	1	Accessory Info Type. See Table 3-46 (page 147)
0x*NN*	4	Accessory capabilities; see Table 3-48 (page 148))
Packet footer specified in "Command Packets" (page 109).		

Table 3-48 Accessory info capabilities bit field

Bit	Capability supported	Accessory requirement	Notes
0	Accessory info capabilities	Required; set to 1	
1	Accessory name	Required; set to 1	Note 1
2	Accessory minimum supported Apple device firmware version	Optional	
3	Accessory minimum supported lingo version	Optional	
4	Accessory firmware version	Required; set to 1	
5	Accessory hardware version	Required; set to 1	
6	Accessory manufacturer	Required; set to 1	Note 1
7	Accessory model number	Required; set to 1	Note 1
8	Accessory serial number	Optional	Note 1
9	Accessory incoming maximum payload size. A packet's payload size is equal to the value of its `length` field, as defined in "Payload Length Field" (page 110).	Optional	
10	Reserved		
11	Accessory status notifications	Optional	
12-17	Reserved		
18	Asynchronous playback state changes	Optional	Note 2
19-31	Reserved		

Notes:

1. The accessory name, manufacturer, model number, and serial number are sent as UTF-8 character arrays and must be less than or equal to 64 bytes (including a null termination character). See Table 3-48 (page 148) for details. Even when this condition is met, the Apple device may not be capable of displaying all the characters in the array in its About box.

2. An accessory should set Bit 18 to 1 if it is able to handle Apple device-driven changes in playback state or playlist contents; see "Playback Status Notifications" (page 59). Such accessories may operate alongside other accessories connected to a single Apple device. All Bluetooth accessories must support this capability.

IMPORTANT: Accessories must provide capability information for all required items listed in Table 3-48 (page 148).

Table 3-49 `RetAccessoryInfo` packet with Accessory Info Type = 0x01/0x06/0x07/0x08

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Accessory Info Type. See Table 3-46 (page 147)
0x*NN*	*NN*	Null-terminated UTF-8 character array
Packet footer specified in "Command Packets" (page 109).		

The `GetAccessoryInfo` packet with Accessory Info Type = 0x02 is deprecated, and is sent to an accessory only by older iPods; nevertheless, the accessory should reply. When the accessory returns the 3-byte minimum supported Apple device firmware major/minor/revision version requested, it must also return the 4-byte Apple device model ID. The accessory should therefore store all the model numbers of older iPods that support it, along with their corresponding minimum supported Apple device firmware versions.

If an unknown or unsupported Apple device model ID is sent to the accessory, the accessory should return the `RetAccessoryInfo` command with the Apple device Model ID and 0xFF as the Apple device firmware major/minor/revision version numbers.

Table 3-50 `RetAccessoryInfo` packet with Accessory Info Type = 0x02 (deprecated)

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x02	1	Accessory Info Type. See Table 3-46 (page 147)
0x*NN*	4	Apple device Model ID
0x*NN*	1	Minimum supported Apple device firmware major version
0x*NN*	1	Minimum supported Apple device firmware minor version
0x*NN*	1	Minimum supported Apple device firmware revision version
Packet footer specified in "Command Packets" (page 109).		

If the accessory's minimum supported Apple device firmware version is higher than the Apple device firmware version, and one or more of the lingo version numbers is higher than that supported by the Apple device, the Apple device will display a message to the user indicating that the Apple device firmware should be updated.

If the accessory's minimum supported Apple device firmware version is smaller than or equal to the Apple device firmware version or any of the major/minor/revision numbers are 0xFF, and one or more of the lingo version numbers is higher than that supported by the Apple device, the Apple device will display a message to the user indicating that the Apple device does not support the accessory.

Table 3-51 `RetAccessoryInfo` packet with Accessory Info Type = 0x03

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x03	1	Accessory Info Type. See Table 3-46 (page 147)
0x*NN*	1	Lingo ID
0x*NN*	1	Major protocol version for lingo ID
0x*NN*	1	Minor protocol version for lingo ID
Packet footer specified in "Command Packets" (page 109).		

Table 3-52 `RetAccessoryInfo` packet with Accessory Info Type = 0x04/0x05

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Accessory Info Type. See Table 3-46 (page 147)
0x*NN*	1	Accessory major version number
0x*NN*	1	Accessory minor version number
0x*NN*	1	Accessory revision version number
Packet footer specified in "Command Packets" (page 109).		

The accessory's incoming maximum packet size indicates the maximum size of a packet from the Apple device that the accessory can support. If the accessory does not return a value, the Apple device assumes that the maximum packet size is 1024 bytes. The maximum packet size must be bigger or equal to 128 bytes and smaller or equal to 65529 (0xFFFA) bytes. A packet's payload size is equal to the value of its `length` field, as defined in "Payload Length Field" (page 110).

Only Apple device packets that contain a UTF-8 character array can have a payload larger than the maximum payload size. In that case, the Apple device will insert a null termination character into the UTF-8 array to force it to fit into the packet. This does not apply to commands that allow multipacket responses.

Note: The maximum payload size restriction does not apply to General lingo packets related to authentication.

Table 3-53 `RetAccessoryInfo` packet with Accessory Info Type = 0x09

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x09	1	Accessory Info Type. See Table 3-46 (page 147)
0x*NN*	2	Max payload size
Packet footer specified in "Command Packets" (page 109).		

Table 3-54 `RetAccessoryInfo` packet with Accessory Info Type = 0x0B

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x0B	1	Accessory Info Type. See Table 3-46 (page 147)
0x*NN*	4	Accessory status types supported; see Table 3-55 (page 151).
Packet footer specified in "Command Packets" (page 109).		

Table 3-55 Accessory status types

Mask bit	Status type	Status description
00	Reserved	
01	0x01	Bluetooth device; see "Bluetooth Autopairing and Connection Status Notifications" (page 55).
02	0x02	Fault condition; see "Command 0x48: AccessoryStatusNotification" (page 180) for fault condition parameters.
31:03	Reserved	

3.3.34 Command 0x29: GetiPodPreferences

Lingo: 0x00 — Origin: Accessory

The accessory sends this command to ask the Apple device to return a specific class of preferences set on it, as defined by a preference class ID. In response, the Apple device sends a `RetiPodPreferences` command.

Table 3-56 `GetiPodPreferences` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Preference class ID (see Table 3-57 (page 152)). Class IDs 0x00-0x02, 0x08-0x0A, and 0x0C-0x0D are available only if the Apple device supports video playback; see "Video Output Settings" (page 47). Class ID 0x03 is available only if the Apple device supports line-out usage.
Packet footer specified in "Command Packets" (page 109).		

Table 3-57 Apple device preference class and setting IDs

Class ID	Preference	Setting	Setting ID	Description
0x00	Video out setting	Off	0x00	Disables Apple device video output of any kind.
		On	0x01	Enables Apple device video output based on the other video preferences (NTSC/PAL, screen configuration, etc.).
		Reserved	0x02–0xFF	
0x01	Screen configuration	Fill entire screen	0x00	Expand the video image to fill the entire screen without letterbox or pillarbox black bars. The "square" pixel proportions are preserved by enlarging both vertical and horizontal dimensions by the same percentage. Depending on the Apple device's media source and video monitor destination aspect ratios, this may result in cutting off the top and bottom or left and right edges of the image; see Note 1, below.
		Fit to screen edge	0x01	Expand the video image to screen edge. The Apple device enlarges the source media so that its top and bottom or left and right edges end at the screen edge without losing any vertical or horizontal image information. The "square" pixel vertical and horizontal proportions are preserved by being enlarged by the same percentage. Depending on the Apple device's media source and video monitor destination aspect ratio, this may result in adding either letterbox black bars at the top and bottom of the screen or pillarbox black bars at the left and right of the screen; see Note 2, below.
		Reserved	0x02–0xFF	
0x02	Video signal format	NTSC	0x00	NTSC video format and timing.
		PAL	0x01	PAL video format and timing.

Class ID	Preference	Setting	Setting ID	Description
		Reserved	0x02-0xFF	
0x03	Line-out usage	Not used	0x00	Line out is not used by accessory and may be disabled by the Apple device. This setting may also indicate that the USB Device mode audio output routing has been selected. See Note 3, below, for requirements and restrictions.
		Used	0x01	Line out is used by accessory and is enabled by the Apple device. This setting may also indicate that the analog line out audio routing has been selected instead of the USB Device mode audio routing. See Note 3, below, for requirements and restrictions.
		Reserved	0x02-0xFF	
0x04-0x07	Reserved			
0x08	Video-out connection	Reserved	0x00	
		Composite	0x01	Composite video connection (interlaced video only).
		S-Video	0x02	S-Video video connection (interlaced video only).
		Component	0x03	Component Y/Pr/Pb video connection (interlaced or progressive scan, depending on the Apple device model).
		Reserved	0x04-0xFF	
0x09	Closed captioning	Off	0x00	Closed caption signaling is disabled.
		On	0x01	The Apple device overlays closed caption text on the video content before outputting the video signal. The text is not present on line 21, so the video monitor cannot do any closed caption processing. **Note:** Closed captions and subtitles cannot be enabled at the same time.
		Reserved	0x02-0xFF	
0x0A	Video monitor aspect ratio (can be set only while video is paused or not playing)	4:3 aspect ratio (fullscreen)	0x00	Used when the video output destination monitor has a horizontal to vertical aspect ratio of 4 to 3, such as an original television monitor format. Depending on the media source format and screen configuration preference, the media content may be displayed fullscreen or have letterbox bars. Apple devices that support widescreen signaling use it to send aspect ratio information to the monitor.

Class ID	Preference	Setting	Setting ID	Description
		16:9 aspect ratio (widescreen)	0x01	Used when the video output destination monitor has a horizontal to vertical aspect ratio of 16 to 9, such as the widescreen theater format. Depending on the media source format and screen configuration preference, the media content may be displayed widescreen or have pillarbox bars. Apple devices that support widescreen signaling use it to send aspect ratio information to the monitor.
		Reserved	0x02-0xFF	
0x0B	Reserved			
0x0C	Subtitles (see Note 4, below)	Subtitles off	0x00	Subtitle overlays are disabled.
		Subtitles on	0x01	The Apple device overlays subtitle text on the video content before outputting the video signal. The video monitor does not need to do any subtitle processing. **Note:** Closed captions and subtitles cannot be enabled at the same time.
		Reserved	0x02-0xFF	
0x0D	Video alternate audio channel	Alternate audio off	0x00	The alternate audio channel is disabled.
		Alternate audio on	0x01	The alternate audio channel is enabled.
		Reserved	0x02-0xFF	
0x0E	Reserved			
0x0F	Pause on power removal	Pause off	0x00	The Apple product does not pause its playback when USB power is removed.
		Pause on	0x01	The Apple product pauses its playback when USB power is removed.
		Reserved	0x02-0xFF	
0x10-0x13	Reserved			
0x14	VoiceOver preference	VoiceOver off	0x00	VoiceOver is disabled.
		VoiceOver on	0x01	VoiceOver is enabled.
		Reserved	0x02-0xFF	
0x15	Reserved			

Class ID	Preference	Setting	Setting ID	Description
0x16	AssistiveTouch	Off	0x00	AssistiveTouch is disabled.
		On	0x01	AssistiveTouch is enabled.
		Reserved	0x02-0xFF	
0x17-0xFF	Reserved			

Notes to Table 3-57 (page 152):

1. If the source media has a 4:3 (fullscreen) aspect ratio and the destination monitor has a 16:9 (widescreen) aspect ratio, the video is enlarged to eliminate the pillarbox bars normally present on the left and right of the screen. This may result in the top and bottom edges of the video being enlarged beyond the limits of the screen and cut off. If the source media has a 16:9 (widescreen) aspect ratio and the destination monitor has a 4:3 (fullscreen) aspect ratio, the video is enlarged to eliminate the letterbox bars normally present on the top and bottom of the screen. This may result in the left and right edges of the video being enlarged beyond the limits of the screen and cut off. See Figure 3-1 (page 156).

2. If the source media has a 4:3 (fullscreen) aspect ratio and the destination monitor has a 16:9 (widescreen) aspect ratio, the video is enlarged so that the image fills the screen vertically, but it may have pillarbox black bars on the left and right edges of the screen. If the source media has a 16:9 (widescreen) aspect ratio and the destination monitor has a 4:3 (fullscreen) aspect ratio, the video is enlarged so that the image fills the screen horizontally, but it may have letterbox black bars on the top and bottom edges of the screen. See Figure 3-1 (page 156).

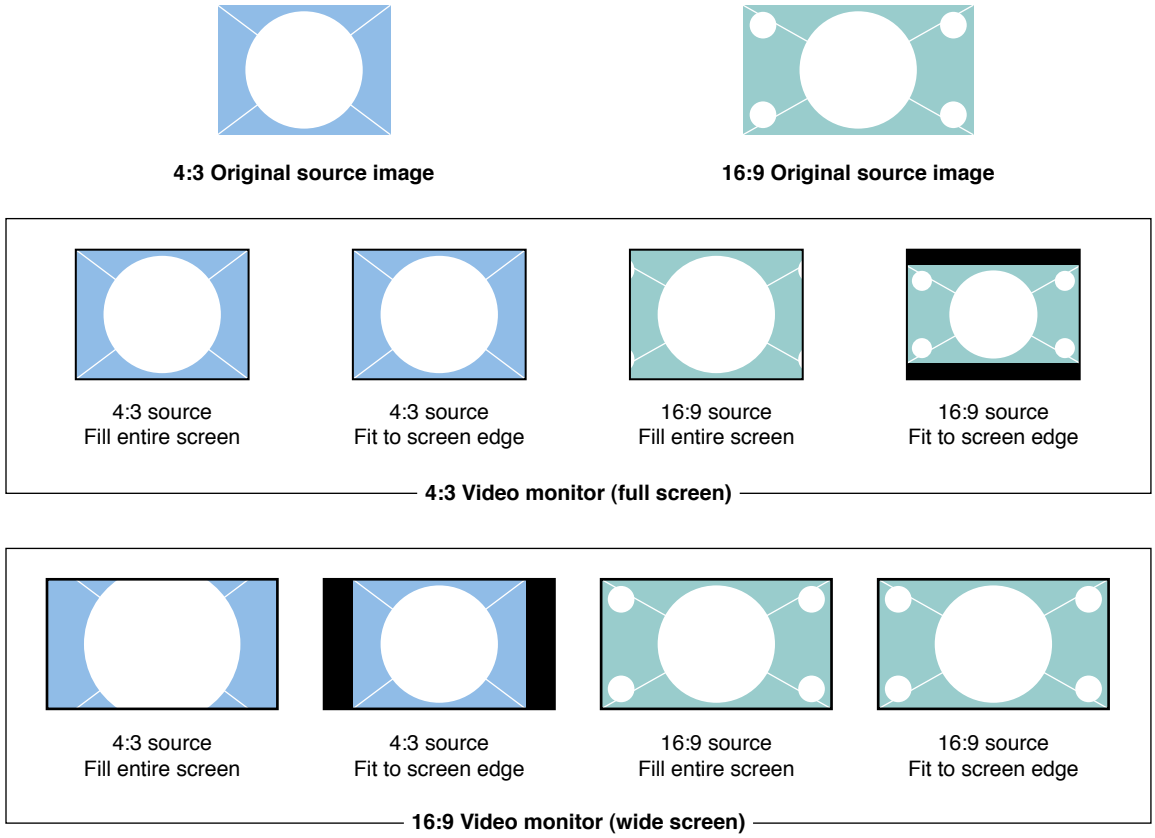
3. The line out usage preference may be used to route Apple device audio to either USB Device mode audio output or analog line out, only if the following are all true:

   - `GetiPodOptionsForLingo` shows that the Apple device supports the General lingo analog/USB audio routing bit
   - Accessory identification includes the Digital Audio lingo
   - The accessory sets the analog/digital audio output routing capabilities bit (bit 21) in its FID `AccessoryCapsToken`.

   The Apple device ignores the `bRestoreOnExit` parameter for this preference, and this parameter should be set to 1. The Apple device will be unconditionally restore it when the accessory is detached.

4. On the iPod touch, subtitle display is not a global setting; it is set for each content item.

Figure 3-1 Screen configuration examples



Authentication Note: To receive video signals from the iPhone and from most Apple devices, and to set their preferences, an accessory must be authenticated; the only class ID that an accessory can get or set without being authenticated is 0x03 (line-out usage). For the best user experience, the Apple device should be configured to the accessory's line out and video out preferences as soon as possible. If an accessory has not set its preferences, the Apple device will make assumptions based on the accessory's Accessory Identify Resistor and declared lingoes. These assumptions may not match the accessory's needs, so it is strongly recommended that accessories set their Apple device preferences promptly. The earliest time at which a accessory can set its video preferences is immediately after the Apple device sends it an `AckAccessoryAuthenticationInfo` command with a value of 0x00 (success status). The 5G iPod is exempt from the authentication requirement.

3.3.35 Command 0x2A: RetiPodPreferences

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command in response to a `GetiPodPreferences` command from the accessory. It echoes the requested preference class ID and the current preference setting for that class.

Table 3-58 `RetiPodPreferences` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Preference class ID (see Table 3-57 (page 152))
0x*NN*	1	Preference setting ID (see Table 3-57 (page 152))
Packet footer specified in "Command Packets" (page 109).		

3.3.36 Command 0x2B: SetiPodPreferences

Lingo: 0x00 — Origin: Accessory

The accessory sends this command to the Apple device to set a specific preference. It sends the preference class ID and the setting ID. If the Restore on Exit field is set to 0x01 the Apple device restores the original setting for this preference when the accessory is disconnected; if it is 0x00, it does not perform the restore. Other values of the Restore on Exit field are illegal.

Note: This command fails if the Apple device does not detect a valid R_{ID} resistor. See "Accessory Detect and Identify" in *MFi Accessory Hardware Specification*.

Although Apple device options need not be used to turn on video output from the Apple device, the Apple device has no default settings. It will reflect the options the user has entered in its settings menu, for which the accessory should not make any assumptions. The following are recommended option-setting practices for accessory designs:

- The accessory should set the video signal format (NTSC or PAL) to ensure signal compatibility with the attached display.
- The accessory should set the video monitor aspect ratio, if possible, to ensure image compatibility with the attached display.
- The accessory should select the audio output path (line out or digital audio out) to ensure that audio will be routed appropriately with its associated video.
- The accessory should always select the Restore on Exit flag to return the Apple device to its previous state when it is disconnected.

Note: Accessories that use the IDPS process must set their video preferences using `SetFIDTokenValues`. Accessories that identify using `IdentifyDeviceLingoes` must set their video preferences after the authentication process has begun and after the accessory has received an `AckAccessoryAuthenticationInfo` command with a value of 0x00. Attempts to set video preferences earlier will result in an `iPodAck` command with an error status. See "Authentication Note" (page 156).

Table 3-59 `SetiPodPreferences` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Preference class ID (see Table 3-57 (page 152))
0x*NN*	1	Preference setting ID (see Table 3-57 (page 152))
0x*NN*	1	Restore on Exit
Packet footer specified in "Command Packets" (page 109).		

Note: The Apple device's line-out state is always restored, regardless of the Restore-on-Exit setting. On some Apple devices, the video-out state is also always restored, regardless of the Restore-on-Exit setting.

3.3.37 Command 0x35: GetUIMode

Lingo: 0x00 — Origin: Accessory

The accessory sends this command to the attached Apple device to determine its current user interface mode. The Apple device replies by sending a `RetUIMode` command.

Table 3-60 `GetUIMode` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

3.3.38 Command 0x36: RetUIMode

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command in response to a `GetUIMode` command, to report its current user interface mode. The modes it may report are listed in Table 3-62 (page 159).

Table 3-61 `RetUIMode` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`UIMode`: The Apple device's current user interface mode; see Table 3-62 (page 159).
Packet footer specified in "Command Packets" (page 109).		

Table 3-62 User interface mode values

Value	Meaning
0x00	Standard Apple device operating mode
0x01	Extended Interface mode; see "Extended Interface Mode" in *iPod Accessory Protocol Interface Specification*.
0x02	iPod Out fullscreen mode
0x03	iPod Out action-safe mode, with video output reduced by 5% on top, bottom, left, and right compared to iPod Out in fullscreen mode. Note that if the action-safe parameter is not available on an older iPod, an Unknown Category iPodAck (status 0x01) is returned to the accessory. See Table 4-262 (page 367).
0x04-0xFF	Reserved

3.3.39 Command 0x37: SetUIMode

Lingo: 0x00 — Origin: Accessory

The accessory sends this command to an Apple device to change its user interface mode. If the Apple device is successful in switching UI modes or is currently in the mode that the `SetUIMode` command requests, it returns a General lingo `iPodAck` command with success status. Otherwise, it returns an `iPodAck` command with the appropriate failure status.

Table 3-63 `SetUIMode` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`UIMode`: The user interface mode to be set; see Table 3-62 (page 159).
Packet footer specified in "Command Packets" (page 109).		

Note: The General Lingo commands previously used to change user interface modes (`RequestExtendedInterfaceMode` (0x03), `ReturnExtendedInterfaceMode` (0x04), `EnterExtendedInterfaceMode` (0x05), and `ExitExtendedInterfaceMode` (0x06)) still work on Apple devices that support iPod Out. Extended Interface mode is available only if the accessory identifies itself successfully for the Extended Interface lingo (Lingo 0x04). `SetUIMode` with `UIMode` set to 0x01 places the Apple device in Extended Interface mode; either `SetUIMode` with `UIMode` set to 0x00 or `ExitExtendedInterfaceMode` places the Apple device in its Standard operating mode. The `EnterExtendedInterfaceMode` command is **deprecated;** it may be used only if the Apple device returns an `iPodAck` command with a Bad Parameter result in response to `SetUIMode`.

3.3.40 Command 0x38: StartIDPS

Lingo: 0x00 — Origin: Accessory

This command is sent by the accessory to start the Identify Device Preferences and Settings (IDPS) process. For details of its use, see "Starting IDPS" (page 96).

Table 3-64 `StartIDPS` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

3.3.41 Command 0x39: SetFIDTokenValues

Lingo: 0x00 — Origin: Accessory

During the IDPS process, the accessory uses this command to send the Apple device a full ID string (FID). This string contains token-value fields that the Apple device is able to parse during the accessory identification process. The Apple device accepts this command only if the accessory previously initiated the IDPS process by sending a `StartIDPS` command.

The accessory may send this command multiple times with different transaction IDs, but it must put as many token-value fields as possible into each command. No iAP command's length-of-packet-payload field may exceed the maximum value determined by calling `RequestTransportMaxPayloadSize`. The accessory must wait for the Apple device to respond to each `SetFIDTokenValues` command with a `AckFIDTokenValues` command before sending the next. If the same token is sent multiple times, the last value will be stored. If a `SetFIDTokenValues` packet is malformed and the Apple device cannot parse one or more token-value fields, the Apple device will responds by sending a General lingo `iPodAck` command with a nonzero status.

Note: The accessory must send certain token-value fields to the Apple device before the accessory can send `EndIDPS` to complete the IDPS process and proceed with authentication. See Table 3-67 (page 161) for these requirements.

Table 3-65 `SetFIDTokenValues` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`numFIDTokenValues`: The number of token-value fields the accessory is setting by this command. If the number of token-value fields the Apple device parses from the command doesn't match this value, the Apple device returns a nonzero `iPodAck` and accepts no token-value fields.
0x*NN*	*NN*	`FIDTokenValues`: Token-value fields containing information the accessory sends to the Apple device before going through authentication. See Table 3-66 (page 161) for the format of each field.
Packet footer specified in "Command Packets" (page 109).		

Table 3-66 `FIDTokenValues` field format

Byte	Size	Name	Description
0	0x01	`length`	Length of this token-value field in bytes, not including this byte.
1	0x01	`FIDType`	First byte of token ID; see Table 3-67 (page 161).
2	0x01	`FIDSubtype`	Second byte of token ID; see Table 3-67 (page 161).
3-*NN*	0x*NN*	`data`	Data corresponding to token; see Table 3-68 (page 163) through Table 3-83 (page 169). All multibyte elements in the `data` field must be big-endian.

Table 3-67 `FIDTokenValues` tokens

`FIDType`	`FIDSubtype`	Token name	Requirements	Format
0x00	0x00	`IdentifyToken`	Required; must be the first token sent.	See Table 3-68 (page 163).
0x00	0x01	`AccessoryCapsToken`	Required	See Table 3-70 (page 163).
0x00	0x02	`AccessoryInfoToken`	Accessory Info Types 0x01, 0x04, 0x05, 0x06, 0x07, and 0x0C required; other types optional. See Table 3-73 (page 165).	See Table 3-72 (page 165).

FIDType	FIDSubtype	Token name	Requirements	Format
0x00	0x03	iPodPreferenceToken	Recommended; required for accessories that use iPod Out mode.	See Table 3-75 (page 166).
0x00	0x04	EAProtocolToken	Required for accessories that communicate with applications.	See Table 3-76 (page 167).
0x00	0x05	BundleSeedIDPref-Token	Required for accessories that communicate with applications.	See Table 3-77 (page 167).
0x00	0x07	ScreenInfoToken	**Deprecated**.	
0x00	0x08	EAProtocolMetadata-Token	Optional; lets accessories declare their preference for matching with applications on the App Store.	See Table 3-80 (page 168).
0x00	0x0E	AccessoryDigital-AudioSampleRates-Token	Optional; recommended for accessories that use USB Device Mode Audio.	See Table 3-82 (page 169).
0x00	0x0F	AccessoryDigital-AudioVideoDelayToken	Optional; recommended for accessories that use USB Device Mode Audio.	See Table 3-83 (page 169).
0x01	0x00	MicrophoneCapsToken	**Deprecated;** do not use in new designs.	

WARNING: The IDPS process in some older Apple devices can handle only IdentifyToken, AccessoryCapsToken, AccessoryInfoToken, iPodPreferenceToken, EAProtocolToken, and BundleSeedIDPrefToken tokens. The accessory can determine this by checking for a value of 0 in bit 23 in the General lingo RetiPodOptionsForLingo option bits; see Table 3-132 (page 192).

The IDPS process will fail unless the Apple device successfully receives all the tokens listed as required in Table 3-67 (page 161). The IdentifyToken token must be sent at the beginning of the first SetFIDTokenValues packet. For the best user experience, the accessory should also return all the recommended tokens that it supports.

The accessory will also fail the IDPS process if it tries to set a preference in the iPodPreferenceToken that requires an accessory capability not previously declared in the AccessoryCapsToken. An example of such a failure would be to declare no line-out support in the AccessoryCapsToken but then try to set a line-out usage preference in the iPodPreferenceToken.

Although an iPodPreferenceToken is not required, Apple devices do not have default settings and the accessory should make no assumptions about their preferences. During the IDPS process, the accessory should set every preference it will require. The accessory can later change Apple device preferences by using the General lingo SetiPodPreferences command.

Table 3-68 `IdentifyToken` format

Name	Bytes	Value	Description
`length`	1	0x*NN*	Length of this token-value field in bytes, not including this byte.
`FIDType`	1	0x00	First byte of token ID.
`FIDSubtype`	1	0x00	Second byte of token ID.
`numLingoes`	1	0x*NN*	Number of bytes in `accessoryLingoes`.
`accessoryLingoes`	*NN*	<var>	One byte for each lingo the accessory supports. For example, 0x00, 0x02, and 0x04 = General lingo + Simple Remote lingo + Extended Interface lingo, with `numLingoes` set to 0x03 and `length` set to 0x0E. The General lingo (0x00) must always be included.
`DeviceOptions`	4	0x*NNNNNNNN*	Accessory's device options; see Table 3-69 (page 163). The accessory must request immediate Authentication 2.0.
`DeviceID`	4	0x*NNNNNNNN*	Accessory's unique identifier, supplied by its iPod Authentication Coprocessor.

Table 3-69 `DeviceOptions` bits in `IdentifyToken`

Bits	Meaning
1:0	Authentication control bits. These bits have the following meanings: 00–01 = reserved 10 = authenticate immediately after identification (required for Authentication 2.0). 11 = reserved
31:2	Reserved; set to 0

Table 3-70 `AccessoryCapsToken` format

Name	Bytes	Value	Description
`length`	1	0x0A	Length of this token-value field in bytes, not including this byte.
`FIDType`	1	0x00	First byte of token ID.
`FIDSubtype`	1	0x01	Second byte of token ID.
`accCapsBitmask`	8	0x*NNNNNNNNNNNNNNNN*	Accessory capabilities; see Table 3-71 (page 164).

Table 3-71 Accessory capabilities bit values

Bit	Accessory capability
00	Analog line-out (accessory consumes Apple device line-out)
01	Analog line-in (accessory supplies line-in to the Apple device)
02	Analog video-out (accessory consumes Apple device video-out)
03	Reserved (set to 0)
04	USB audio-out (accessory consumes digital audio from Apple device in USB Device mode)
05-08	Reserved (set to 0)
09	Accessory supports communication with an application.
10	Reserved (set to 0)
11	Accessory checks Apple device volume, either by registering for Display Remote lingo notifications or by sending `GetiPodStateInfo` commands with an `infoType` of 0x04 or 0x10.
12–16	Reserved (set to 0)
17	Accessory uses Apple device user interface VoiceOver features; see "VoiceOver" (page 81).
18	The accessory can safely handle asynchronous playback state changes from the Apple device, even when the accessory did not initiate those changes; see "Playback Status Notifications" (page 59). Such accessories may operate alongside other accessories connected to a single Apple device. All Bluetooth accessories must support this capability.
19	Accessory can handle a multi-packet response from the Apple device, even when it is mixed with other packets that may include notifications and responses to other commands. This bit must be set for an accessory to cancel long data downloads; see "Command 0x50: CancelCommand" (page 203). See "Interleaved Transfers" (page 117).
20	Reserved (set to 0)
21	1 = To maintain digital audio/video synchronization when playing video media, the USB host accessory will switch between USB Device mode audio and analog line out audio routings when playing video and non-video media. This bit may be set only if these conditions are true: ■ The accessory is a USB host and the Apple device is a USB device, and the accessory identifies itself for the Digital Audio lingo. ■ The Digital Audio lingo does not support the `SetVideoDelay` command; see `GetiPodOptionsForLingo` for the Digital Audio lingo. ■ The General lingo supports the analog/USB audio routing feature using the `SetiPodPreferences` line out preference; see `GetiPodOptionsForLingo` for the General lingo. 0 = The accessory as a USB host will not switch audio routings when playing video and non-video media using the USB Device mode audio routing. If the accessory has identified itself for the Digital Audio lingo, sending a `SetiPodPreferences` command with the line out class ID has no effect.

Bit	Accessory capability
22	Reserved (set to 0)
23	Accessory can support AssistiveTouch cursor feature; see "AssistiveTouch" (page 82).
24–63	Reserved (set to 0)

Table 3-72 `AccessoryInfoToken` format

Name	Bytes	Value	Description
`length`	1	0x*NN*	Length of this token-value field in bytes, not including this byte.
`FIDType`	1	0x00	First byte of token ID.
`FIDSubtype`	1	0x02	Second byte of token ID.
`accInfoType`	1	0x*NN*	Accessory info type; see Table 3-73 (page 165).
`accInfo`	*NN*	<var>	Accessory info; see Table 3-73 (page 165).

Table 3-73 Accessory Info Type values

Value	Meaning	Parameter
0x00	Reserved	
0x01	Accessory name	Null-terminated UTF-8 string (64 bytes maximum, including terminator)
0x02	Reserved	
0x03	Reserved	
0x04	Accessory firmware version	3 bytes
0x05	Accessory hardware version	3 bytes
0x06	Accessory manufacturer	Null-terminated UTF-8 string (64 bytes maximum, including terminator)
0x07	Accessory model number	Null-terminated UTF-8 string (64 bytes maximum, including terminator)
0x08	Accessory serial number	Null-terminated UTF-8 string (64 bytes maximum, including terminator)
0x09	Accessory incoming maximum payload size	2 bytes. The maximum payload size must be larger or equal to 128 bytes and smaller or equal to 65529 (0xFFFA) bytes.
0x0A	Reserved	
0x0B	Accessory status	32-bit mask; see Table 3-55 (page 151).

Value	Meaning	Parameter
0x0C	Accessory RF certifications	32-bit `RFCertificationDeclaration` (RF certification mask); see Table 3-74 (page 166). Each bit may be set only if the accessory has been tested by an Apple-authorized independent laboratory and has passed all the RF tests required for that class. Otherwise all bits must be 0. See Apple's *MFi Accessory Testing Specification* for details of the tests required.
0x0D–0xFF	Reserved	

Table 3-74 Accessory RF certification declarations

Mask bit	Certification	Products
00	Class 1	iPhone, iPhone 3G, iPhone 3GS
01	Class 2	iPhone 4 (GSM model)
02	Class 3	Reserved
03	Class 4	iPhone 4 (CDMA model)
04	Class 5	iPhone 4S
05	Class 6	iPhone 5 (A1428 model)
06	Class 7	iPhone 5 (A1429 model)
07:31	Reserved	

Table 3-75 `iPodPreferenceToken` format

Name	Bytes	Value	Description
`length`	1	0x05	Length of this token-value field in bytes, not including this byte.
`FIDType`	1	0x00	First byte of token ID.
`FIDSubtype`	1	0x03	Second byte of token ID.
`iPodPrefClass`	1	0x*NN*	Apple device preference class ID; see Table 3-57 (page 152).
`prefClassSetting`	1	0x*NN*	Apple device preference setting ID; see Table 3-57 (page 152).
`restoreOnExit`	1	0x01	Must be set to 0x01.

Table 3-76 `EAProtocolToken` format

Name	Bytes	Value	Description
`length`	1	0x*NN*	Length of this token-value field in bytes, not including this byte.
`FIDType`	1	0x00	First byte of token ID.
`FIDSubtype`	1	0x04	Second byte of token ID.
`protocolIndex`	1	0x*NN*	A unique protocol index, starting from 1. Subsequent indexes in other `EAProtocolToken` token-value fields must increment by 1.
`protocolString`	*NN*	<var>	A null-terminated UTF-8 reverse-DNS string (e.g. "com.acme.gadget"). This string will be compared (without considering case) to strings presented by applications on the iPhone or iPod touch.

Note: If an accessory sends a duplicate `protocolString` or repeats a `protocolIndex` number, the last received `EAProtocolToken` will be used; any information from a previous `EAProtocolToken` will be overwritten.

Table 3-77 `BundleSeedIDPrefToken` format

Name	Bytes	Value	Description
`length`	1	0x0D	Length of this token-value field in bytes, not including this byte.
`FIDType`	1	0x00	First byte of token ID.
`FIDSubtype`	1	0x05	Second byte of token ID.
`BundleSeedIDString`	11	<var>	A null-terminated UTF-8 string (e.g. "A1B2C3D4E5") that identifies the vendor of a preferred application, with case sensitivity. This string is derived from the vendor's App ID assigned by Apple.

Table 3-78 `ScreenInfoToken` format (deprecated)

Name	Bytes	Value	Description
`length`	1	0x10	Length of this token-value field in bytes, not including this byte.
`FIDType`	1	0x00	First byte of token ID.
`FIDSubtype`	1	0x07	Second byte of token ID.
`totalScreenWidthInches`	2	0x*NN*	Accessory's approximate screen width in inches
`totalScreenHeightInches`	2	0x*NN*	Accessory's approximate screen height in inches
`totalScreenWidthPixels`	2	0x*NN*	Number of pixels along the accessory's screen width

Name	Bytes	Value	Description
totalScreenHeightPixels	2	0x*NN*	Number of pixels along the accessory's screen height
iPodOutScreenWidthPixels	2	0x*NN*	Number of pixels along the iPod Out screen width allotment
iPodOutScreen-HeightPixels	2	0x*NN*	Number of pixels along the iPod Out screen height allotment
screenFeaturesMask	1	0x*NN*	A bitmask with bits set to represent the accessory screen's features; see Table 3-79 (page 168).
screenGammaValue	1	0x*NN*	A representation of the accessory display's gamma value. The Apple device obtains the gamma value from this byte by dividing it by 100 and adding 1.00. For example, a gamma of 2.200 is represented by a screenGammaValue of 120.

Table 3-79 Accessory screen features

Bit	Meaning
00	The accessory has a color display
07-01	Reserved

Table 3-80 EAProtocolMetadataToken format

Name	Bytes	Value	Description
length	1	0x04	Length of this token-value field in bytes, not including this byte.
FIDType	1	0x00	First byte of token ID.
FIDSubtype	1	0x08	Second byte of token ID.
protocolIndex	1	0x*NN*	A unique protocol index, starting from 1. Subsequent indexes in other EAProtocolToken token-value fields must increment by 1.
metadataType	1	0x*NN*	See Table 3-81 (page 168).

Table 3-81 MetadataType values

Value	Meaning
0x00	The Apple device does not try to find a matching app. It does not display a "Find App For This Accessory" button in the Settings > General > About > Accessory Name screen.
0x01	The Apple device tries to find a matching app. It displays a "Find App For This Accessory" button in the Settings > General > About > Accessory Name screen.

Value	Meaning
0x02	Reserved.
0x03	The Apple device does not try to find a matching app, but it displays a "Find App For This Accessory" button in the Settings > General > About > Accessory Name screen so the user can find one.
0x04-0xFF	Reserved.

Table 3-82 `AccessoryDigitalAudioSampleRatesToken` format

Name	Bytes	Value	Description
`length`	1	0x*NN*	Length of this token-value field in bytes, not including this byte.
`FIDType`	1	0x00	First byte of token ID.
`FIDSubtype`	1	0x0E	Second byte of token ID.
`SampleRates`	0x*NN*	0x*NN*	Supported sample rates as a big endian list of 32-bit integers, one for each supported sample rate.

Table 3-83 `AccessoryDigitalAudioVideoDelayToken` format

Name	Bytes	Value	Description
`length`	1	0x06	Length of this token-value field in bytes, not including this byte.
`FIDType`	1	0x00	First byte of token ID.
`FIDSubtype`	1	0x0F	Second byte of token ID.
`VideoDelayInMs`	0x04	0x*NN*	Video synchronization delay in milliseconds.

3.3.42 Command 0x3A: AckFIDTokenValues

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command in response to each `SetFIDTokenValues` packet from the accessory. The number of token-value fields contained in `FIDTokenValueACKs` matches exactly the number of fields the accessory sent in its `SetFIDTokenValues` command. The different statuses returned to the accessory let it determine whether to retry a failed token-value field.

Table 3-84 `AckFIDTokenValues` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		

Value	Bytes	Parameter
0x*NN*	1	`numFIDTokenValueACKs`: The number of token-value fields the Apple device is acknowledging. This number matches the number the accessory sent in the `SetFIDTokenValues` command.
0x*NN*	*NN*	`FIDTokenValueACKs`: Acknowledgement values for the token-value fields sent by `SetFIDTokenValues`. See Table 3-85 (page 170) through Table 3-95 (page 173).
Packet footer specified in "Command Packets" (page 109).		

Table 3-85 Acknowledgment format for `IdentifyToken`

Name	Bytes	Value	Description
`length`	1	0x03	Length of this token-value field in bytes, not including this byte.
`FIDType`	1	0x00	First byte of token ID.
`FIDSubtype`	1	0x00	Second byte of token ID.
`ACKStatus`	1	0x*NN*	Status of acknowledgment; see Table 3-86 (page 170).
`lingoIDs`	0x*NN*	0x*NN*	If `ACKStatus` = 4, IDs of busy lingoes; see Table 3-86 (page 170).

Table 3-86 Acknowledgment status codes

Value	Description
0	Token-value field accepted.
1	Required token-value field failed.
2	Optional token-value field recognized, but failed; the accessory may retry, either with another `SetFIDTokenValues` command while in the IDPS process or with another iAP command that emulates the desired setting. The accessory should not allow this status return to prevent it from completing IDPS process.
3	Token not supported; the accessory must not retry with this Apple device.
4	Lingoes busy. This status responds to an `IdentifyToken` that tries to register for lingoes that are busy on another port. The ID numbers of the busy lingoes are appended to the token. See `lingoIDs` in Table 3-85 (page 170).
5	Maximum connections reached. This status responds to an `IdentifyToken` if another accessory is connected and the calling accessory cannot connect to the Apple device.
6–255	Reserved

Table 3-87 Acknowledgment format for `AccessoryCapsToken`

Name	Bytes	Value	Description
`length`	1	0x03	Length of this token-value field in bytes, not including this byte.
`FIDType`	1	0x00	First byte of token ID.
`FIDSubtype`	1	0x01	Second byte of token ID.
`ACKStatus`	1	0x*NN*	Status of acknowledgment; see Table 3-86 (page 170).

Table 3-88 Acknowledgment format for `AccessoryInfoToken`

Name	Bytes	Value	Description
`length`	1	0x04	Length of this token-value field in bytes, not including this byte.
`FIDType`	1	0x00	First byte of token ID.
`FIDSubtype`	1	0x02	Second byte of token ID.
`ACKStatus`	1	0x*NN*	Status of acknowledgment; see Table 3-86 (page 170).
`accInfoType`	1	0x*NN*	Accessory info type in token being acknowledged.

Table 3-89 Acknowledgment format for `iPodPreferenceToken`

Name	Bytes	Value	Description
`length`	1	0x04	Length of this token-value field in bytes, not including this byte.
`FIDType`	1	0x00	First byte of token ID.
`FIDSubtype`	1	0x03	Second byte of token ID.
`ACKStatus`	1	0x*NN*	Status of acknowledgment; see Table 3-86 (page 170).
`iPodPrefClass`	1	0x*NN*	Apple device preference class in token being acknowledged.

Table 3-90 Acknowledgment format for `EAProtocolToken`

Name	Bytes	Value	Description
`length`	1	0x04	Length of this token-value field in bytes, not including this byte.
`FIDType`	1	0x00	First byte of token ID.
`FIDSubtype`	1	0x04	Second byte of token ID.
`ACKStatus`	1	0x*NN*	Status of acknowledgment; see Table 3-86 (page 170).
`protocolIndex`	1	0x*NN*	Protocol index in token being acknowledged.

Table 3-91 Acknowledgment format for `BundleSeedIDPrefToken`

Name	Bytes	Value	Description
`length`	1	0x03	Length of this token-value field in bytes, not including this byte.
`FIDType`	1	0x00	First byte of token ID.
`FIDSubtype`	1	0x05	Second byte of token ID.
`ACKStatus`	1	0x*NN*	Status of acknowledgment; see Table 3-86 (page 170).

Table 3-92 Acknowledgment format for `ScreenInfoToken` (deprecated)

Name	Bytes	Value	Description
`length`	1	0x03	Length of this token-value field in bytes, not including this byte.
`FIDType`	1	0x00	First byte of token ID.
`FIDSubtype`	1	0x07	Second byte of token ID.
`ACKStatus`	1	0x*NN*	Status of acknowledgment; see Table 3-86 (page 170).

Table 3-93 Acknowledgment format for `EAProtocolMetadataToken`

Name	Bytes	Value	Description
`length`	1	0x03	Length of this token-value field in bytes, not including this byte.
`FIDType`	1	0x00	First byte of token ID.
`FIDSubtype`	1	0x08	Second byte of token ID.
`ACKStatus`	1	0x*NN*	Status of acknowledgment; see Table 3-86 (page 170).

Table 3-94 Acknowledgment format for `AccessoryDigitalAudioSampleRatesToken`

Name	Bytes	Value	Description
`length`	1	0x03	Length of this token-value field in bytes, not including this byte.
`FIDType`	1	0x00	First byte of token ID.
`FIDSubtype`	1	0x0E	Second byte of token ID.
`ACKStatus`	1	0x*NN*	Status of acknowledgment; see Table 3-86 (page 170).

Table 3-95 Acknowledgment format for `AccessoryDigitalAudioVideoDelayToken`

Name	Bytes	Value	Description
`length`	1	0x03	Length of this token-value field in bytes, not including this byte.
`FIDType`	1	0x00	First byte of token ID.
`FIDSubtype`	1	0x0F	Second byte of token ID.
`ACKStatus`	1	0x*NN*	Status of acknowledgment; see Table 3-86 (page 170).

3.3.43 Command 0x3B: EndIDPS

Lingo: 0x00 — Origin: Accessory

This command is sent by the accessory to end the IDPS process. The Apple device takes different actions depending on the value of `accIDPSStatus`.

If the accessory sends this command while the Apple device is not in the IDPS process, the Apple device responds with an `iPodAck` command that passes a nonzero status. If the Apple device is in the IDPS process and the accessory sends this command with an unsupported `accEndIDPSStatus` value, the Apple device remains in the IDPS process.

Table 3-96 `EndIDPS` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`accEndIDPSStatus`: The accessory's IDPS status, which determines what action the Apple device takes after receiving `EndIDPS`. See Table 3-97 (page 173).
Packet footer specified in "Command Packets" (page 109).		

Table 3-97 `accEndIDPSStatus` values

Value	Actions to be taken
0	The accessory has finished with IDPS, and asks the Apple device to continue with authentication if the Apple device sends a `IDPSStatus` command with success status.
1	The accessory asks to reset all IDPS information it has sent to the Apple device. The accessory must then send another `StartIDPS` command and successfully complete the IDPS process within the time limit specified in "Command 0x38: StartIDPS" (page 160).
2	The accessory has determined that the Apple device doesn't support features the accessory needs. The accessory is abandoning the IDPS process.
3	The accessory has finished with IDPS on the current transport and is restarting IDPS on another transport. The accessory must cease traffic on the old transport immediately and restart IDPS on the new transport within 2 seconds.

Value	Actions to be taken
4–255	Reserved

Note: In the future, an accessory may be connected to an Apple device that supports IDPS but doesn't recognize a specific token-value field. To handle this case, the accessory must respond to the `accIDPSStatus` value of 2. For example, if the accessory needs to set a token-value field that the attached Apple device cannot parse, it may determine that it can't work with the Apple device and hence must discontinue the identification process.

3.3.44 Command 0x3C: IDPSStatus

Lingo: 0x00 — Origin: Apple device

This command is sent by the Apple device after an accessory sends `EndIDPS`. The Apple device determines whether all required token-value fields were successfully sent by the accessory, and sends a `status` value that indicates whether authentication will proceed and the Apple device will apply the tokens, or the accessory failed to identify itself properly.

Note: After receiving an EndIDPS with status 0, the Apple device evaluates the set of FID tokens that the accessory sent for completeness and consistency. At this point the accessory will fail the IDPS process if it has set a preference in the `iPodPreferenceToken` that requires an accessory capability not declared in the `AccessoryCapsToken`. An example of such an inconsistency would be to declare no line-out support in the `AccessoryCapsToken` but also set a line-out usage preference in the `iPodPreferenceToken`.

Table 3-98 `IDPSStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`status`: A value that indicates whether the accessory successfully identified during the IDPS sequence. Combined with the value of `accEndIDPSStatus` sent by the accessory, it determines the Apple device's action. See Table 3-99 (page 174).
Packet footer specified in "Command Packets" (page 109).		

Table 3-99 Apple device actions in response to `accEndIDPSStatus` and IDPS status

accEndIDPSStatus	IDPS status	Action taken by the Apple device
0	0	The Apple device received all required token-value fields; authentication will proceed. Optional token-value fields may or may not have been received, but their status will not block authentication.

accEndIDPSStatus	IDPS status	Action taken by the Apple device
	1	One or more required token-value fields were rejected by an FIDTokenValueACK command and were not correctly resent; IDPS fails.
	2	One or more required token-value fields were missing; IDPS fails.
	3	One or more required token-value fields were rejected by an FIDTokenValueACK command and were not correctly resent, plus one or more required token-value fields were missing; IDPS fails.
1	4	The IDPS time limit was not exceeded; the accessory may retry IDPS or send IdentifyDeviceLingoes.
	5	The IDPS time limit was reached; the accessory cannot retry IDPS but may send IdentifyDeviceLingoes.
2	6	The Apple device will not accept any token-value fields and will not continue with authentication. The accessory may send IdentifyDeviceLingoes but not StartIDPS.
0	7	IDPS fails for one or more of the following reasons: ■ The accessory sent an Apple device Preference Token (iPodPreferenceToken) for preference class 0x03 (line-out) without indicating line-out support in the accCapsBitmask field of its Accessory Capabilities Token (AccessoryCapsToken). ■ The accessory sent an Apple device Preference Token (iPodPreferenceToken) for at least one of preference classes 0x00 (video-out), 0x01 (video-out screen configuration), 0x02 (video-out signal format), 0x08 (video-out connection), or 0x0A (video monitor aspect ratio) without indicating video-out support in the accCapsBitmask field of its Accessory Capabilities Token (AccessoryCapsToken). ■ The accessory set any EA Protocol Tokens (EAProtocolToken), or a Preferred Bundle Seed ID (BundleSeedIDPrefToken), without setting bit 09 in its Accessory Capabilities Token (AccessoryCapsToken). ■ The accessory identified for the Digital Audio Lingo in its (IdentifyToken) without setting the USB Audio bit (bit 04) in its Accessory Capabilities Token (AccessoryCapsToken).
NN	8–255	Reserved

Note: When a failed IDPS process is retried, the Apple device does not retain any successfully set token-value fields from the failed IDPS process. Accessories must send all token-value fields again.

3.3.45 Command 0x3F: OpenDataSessionForProtocol

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command to tell the accessory that a data stream has been opened between the accessory and an iOS application. It sends the accessory the `sessionID` of the data stream and the `protocolIndex` to which the open session corresponds. All communications between the accessory and the application for the given `sessionID` are assumed to use the protocol specified by `protocolIndex`.

The accessory must send an `AccessoryAck` command in response. A success status means the accessory can support a new open data stream. If the accessory sends an `AccessoryAck` command with other than success status, the Apple device informs the application that the accessory has refused the data stream connection.

Table 3-100 `OpenDataSessionForProtocol` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	2	`sessionID`: Session ID for subsequent data transfer commands
0x*NN*	1	`protocolIndex`: Index of the protocol for which this session is being opened; see Table 3-76 (page 167).
Packet footer specified in "Command Packets" (page 109).		

3.3.46 Command 0x40: CloseDataSession

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command to tell the accessory that the data stream on the given `sessionID` is closing, ending communication. The accessory must send an `AccessoryAck` response, but the data stream will remain closed regardless of the `AccessoryAck` status. If the accessory sends any more `AccessoryDataTransfer` packets for the closed session they will be acknowledged with an Unknown Category status (0x01). The Apple device never retries a `CloseDataSession` command.

Table 3-101 `CloseDataSession` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	2	`sessionID`: Session ID of the session being closed
Packet footer specified in "Command Packets" (page 109).		

3.3.47 Command 0x41: AccessoryAck

Lingo: 0x00 — Origin: Accessory

The accessory must use this command to acknowledge General lingo commands 0x3F (`OpenDataSessionForProtocol`), 0x40 (`CloseDataSession`), and 0x43 (`iPodDataTransfer`) from the Apple device. This command must not be used for any other purpose.

Table 3-102 `AccessoryAck` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`ackStatus`: Status of command being acknowledged: must be either 0x00 (Success) or 0x04 (Bad Parameter).
0x*NN*	1	`cmdID`: ID of command being acknowledged; must be 0x3F, 0x40, or 0x43.
Packet footer specified in "Command Packets" (page 109).		

3.3.48 Command 0x42: AccessoryDataTransfer

Lingo: 0x00 — Origin: Accessory

The accessory uses this command to send data to an application listening for a specific `sessionID`, as established by a prior `OpenDataSessionForProtocol` command. The Apple device responds with a General lingo `iPodAck` command indicating whether or not the application was able to receive the data from this command.

Note: If the accessory sends an `AccessoryDataTransfer` command for a session that is closed or doesn't exist, the Apple device acknowledges it with an Unknown Category status (0x01). If the accessory sends an `AccessoryDataTransfer` command for a session that the Apple device has opened but which the accessory has not acknowledged successfully, the Apple device acknowledges it with a Command Failed status (0x02).

During a communication session, the following rules apply to `AccessoryDataTransfer` commands:

- The accessory should use `AccessoryDataTransfer` commands whenever it has data it wants to stream to an application with an open session.
- The accessory must use an appropriate iAP packet size, as specified in "Command Packets" (page 109).
- The accessory must not send a new `AccessoryDataTransfer` packet until the previous one has been acknowledged (as successful or not) or a 500 ms timeout has expired.
- Upon receiving a successful acknowledgment of an `AccessoryDataTransfer` command, the accessory may immediately send a new command containing available data for any session ID.
- If a 500 ms timeout occurs before an `AccessoryDataTransfer` command is acknowledged successfully, the accessory may resend the same packet with the same transaction ID. If the accessory retries 10 times without receiving a successful acknowledgement, it must close the session and assume that the transfer failed.
- If an `AccessoryDataTransfer` command is acknowledged with a `SessionWriteFailure` status (0x17), the accessory may send a packet with the same transaction ID containing the remaining data. If no additional bytes are successfully accepted after 10 such attempts, the accessory must close the session and assume that the transfer failed. See Table 3-9 (page 127).
- If the accessory receives an `iPodNotification` command for flow control, it must not send any packets (including retries of `AccessoryDataTransfer` packets) during the specified waiting time.

The accessory can choose any packet format specified in "Command Packets" (page 109), and it can determine the maximum payload size that it can deliver in a single packet by sending a `RequestTransportMaxPayloadSize` command.

Table 3-103 `AccessoryDataTransfer` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	2	`sessionID`: Session ID of the connection between the application and the accessory
0x*NN*	*NN*	`data`: Data the accessory sends to the listening application.
Packet footer specified in "Command Packets" (page 109).		

3.3.49 Command 0x43: iPodDataTransfer

Lingo: 0x00 — Origin: Apple device

The Apple device uses this command to send data to an accessory listening for a specific `sessionID`, as established by a prior `OpenDataSessionForProtocol` command. The accessory must respond with an `AccessoryAck` command that passes a command ID of 0x43 and the same transaction ID as contained in the `iPodDataTransfer` packet.

Note: The Apple device will not send this command until accessory authentication has finished. The Authentication 2.0B process may last up to 2 seconds after accessory identification.

By default, the maximum `iPodDataTransfer` packet payload length is 1,018 bytes. The accessory can specify a different maximum length by sending an `AccessoryInfoToken` with an `accInfoType` of 0x09. The length that can be set ranges from 128 to 65,529 bytes; specifying less than 128 bytes will revert to 128. The packet payload length is equal to the value of the `length` field defined in "Payload Length Field" (page 110).

The Apple device may send data in any format defined in "Command Packets" (page 109). Accessories must be prepared to receive packets of any size unless they have specified a maximum packet payload length through a `RetAccessoryInfo` command.

During a communication session, the following rules apply to `iPodDataTransfer` commands:

- The Apple device will send an initial `iPodDataTransfer` packet when an application has placed data into the accessory's input queue.
- Upon receiving an acknowledgment of an `iPodDataTransfer` command, the Apple device may immediately send a new command containing available data for any session ID.
- The accessory's acknowledgment of an `iPodDataTransfer` command with Success status (0x00) must signify that the packet has been received, it has been moved out of the accessory's lowest-level receive queue, and that the accessory is ready to receive another packet.

- The only circumstance under which an accessory may acknowledge an `iPodDataTransfer` command with a nonzero status is if no prior `OpenDataSessionForProtocol` command has established the command's session ID. In this case it must send a Bad Parameter (0x04) status, and the Apple device may close the session.
- If an `iPodDataTransfer` command is acknowledged with an error status, the Apple device may optionally purge its outbound packet queue of all pending `iPodDataTransfer` packets for that session ID. The Apple device will not otherwise discard any data in a session.
- If an accessory's data receive queue cannot accept another packet, then the accessory must not acknowledge the `iPodDataTransfer` command until packet acceptance is restored. This may cause the Apple device to retry the `iPodDataTransfer` command up to ten times. If the Apple device receives no acknowledgment after the tenth retry, it may close the session.
- If a 500 ms timeout occurs before an `iPodDataTransfer` command is acknowledged, the Apple device will resend the same packet with the same transaction ID.

Table 3-104 `iPodDataTransfer` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	2	`sessionID`: Session ID of the connection between the application and the accessory
0x*NN*	*NN*	`data`: Data the application sends to the listening accessory.
Packet footer specified in "Command Packets" (page 109).		

3.3.50 Command 0x46: SetAccessoryStatusNotification

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command to the accessory to request status notifications for specific status types. The accessory must respond with a `RetAccessoryStatusNotification` command, acknowledging the settings.

Immediately after sending a `RetAccessoryStatusNotification` command, the accessory must send an initial `AccessoryStatusNotification` command for each notification for which the Apple device has registered, whether or not the corresponding status has changed.

The accessory must then send an `AccessoryStatusNotification` command for each status type set every time that status changes. Notifications should not be sent too frequently and notifications for the same status type must be sent no more often than once every 500 ms.

Table 3-105 `SetAccessoryStatusNotification` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x00	4	Status notifications mask; see Table 3-55 (page 151).

Value	Bytes	Parameter
Packet footer specified in "Command Packets" (page 109).		

Table 3-110 (page 182) lists a sample command flow that uses the General lingo 0x46–0x48 accessory status notification commands.

3.3.51 Command 0x47: RetAccessoryStatusNotification

Lingo: 0x00 — Origin: Accessory

The accessory sends this command to the Apple device to acknowledge receipt of a `SetAccessoryStatusNotification` command. Bits in the status notification mask must be set for each notification type successfully enabled in the accessory.

Table 3-106 `RetAccessoryStatusNotification` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x00	4	Status notifications mask; see Table 3-55 (page 151).
Packet footer specified in "Command Packets" (page 109).		

Table 3-110 (page 182) lists a sample command flow that uses the General lingo 0x46–0x48 accessory status notification commands.

3.3.52 Command 0x48: AccessoryStatusNotification

Lingo: 0x00 — Origin: Accessory

The accessory must send this command to the Apple device when there is a change in one of the status types for which the Apple device registered by previously sending a `SetAccessoryStatusNotification` command. Notifications should not be sent too frequently and notifications for the same status type must be sent no more often than once every 500 ms.

Immediately after sending a `RetAccessoryStatusNotification` command, the accessory must send an initial `AccessoryStatusNotification` command for each notification for which the Apple device registered, regardless of whether the corresponding status has changed.

Table 3-107 `AccessoryStatusNotification` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Status type; see Table 3-55 (page 151).

Value	Bytes	Parameter
0x*NN*	*NN*	Status parameters; see Table 3-108 (page 181).
Packet footer specified in "Command Packets" (page 109).		

Table 3-108 `AccessoryStatusNotification` parameters

Status type	Parameters	Description
0x01	8–*NN*	Concatenation of one or more 36-byte fields, each describing a Bluetooth device for autopairing; see "Bluetooth Autopairing and Connection Status Notifications" (page 55) and Table 3-109 (page 181).
0x02	8	Accessory fault type (see **Note** below): 0x00: No fault 0x01: Voltage fault 0x02: Current fault 0x03–0xFF: Reserved
	9	Accessory fault condition: 0x00: No fault 0x01: Recoverable 0x02: Not recoverable 0x03–0xFF: Reserved

Note: When an accessory sends an `AccessoryStatusNotification` command for an accessory fault it must pass both a fault type and a fault condition.

Table 3-109 Bluetooth device description

Bytes	Name	Description
0-3	`devType`	32-bit Device Type: Bit 00: Bluetooth device type Classic Bits 01-31: Reserved
4-7	`devState`	32-bit Device State: Bit 00: Bluetooth device type Classic on/off bit; 0 = off, 1 = on Bits 01-31: Reserved
8-11	`devClass`	A parameter indicating the type of device and the types of service that are supported, as described in *Bluetooth Core Specification Version 4.0,* Volume 3, Section 3.2.4.

Bytes	Name	Description
12	`pairingType`	8-bit Pairing Type: Bit 00: None Bit 01: PIN code Bit 02: Secure Simple Pairing (SSP) Bits 03-07: Reserved
13	Reserved.	Set to 0.
14-29	PIN code	A null-terminated UTF-8 string (compatible with devices before Bluetooth 2.1)
30-35	MAC address	Media Access Control address

Table 3-110 (page 182) lists a sample command flow that uses the General lingo 0x46–0x48 accessory status notification commands.

Table 3-110 Sample IDPS process and accessory status notification commands

Step	Accessory command	Apple device command	Comment
1	`StartIDPS`		no params
2		`iPodAck`	acknowledging 'Success (OK)' to command 'General Lingo::StartIDPS'

If the `iPodAck` reply to `StartIDPS` returns a status of 0x00, the accessory proceeds to Step 3. Otherwise,

- If the accessory receives no `iPodAck` command within 500 ms, it may retry Step 1 at 500 ms intervals as long as Apple Device Detect (pin 30 of the 30-pin connector) is low and Accessory Power (pin 13) is high. Accessories that don't monitor these signals (such as Bluetooth accessories or USB accessories without a 191 kΩ ID resistor) may retry without checking them.
- If the accessory receives an `iPodAck` status of 0x04 (Bad Parameter), the Apple device does not support IDPS. The accessory must not proceed to Step 3. Instead, it must perform either one of these two alternative actions:
  - If IDPS support is required, within 800 ms the accessory must send an `IdentifyDeviceLingoes` command with all fields set to 0xFF. This causes the Apple device to display an "Accessory not supported" message.
  - If IDPS support is not required, the accessory must then send another `IdentifyDeviceLingoes` command with its lingo mask set for only the General lingo, with no options, and with a Device ID of 0x00. See "Canceling a Current Authentication Process With IdentifyDeviceLingoes" (page 135). A sample command sequence is listed in Table C-48 (page 553).
- If the accessory receives any other nonzero `iPodAck` status, it must not send any more iAP commands until it has been disconnected. Upon reconnection it must restart identification at Step 1.

Step	Accessory command	Apple device command	Comment
3	SetFIDTokenValues		setting FID tokens: IdentifyToken: General Lingo 00 device options: 02 (auth immediately); device id: 0200. AccessoryCapsToken: line out. AccessoryInfoToken: info type 04, Accessory Firmware Version: 1.0.0; info type 05, Accessory Hardware Version: 1.0.0; info type 06, Accessory Manufacturer: Apple; info type 07, Accessory Model Number: 1234; info type 08, Accessory Serial Number: 5678; info type 01, Accessory Name = Brabazon: 42 72 61 62 61 7A 6F 6E 00; info type 0B, Accessory Status Types supported: 00 00 00 02.
4		AckFIDTokenValues	returning acknowledgments for FID tokens
5	EndIDPS		terminating IDPS process
6		IDPSStatus	status 'ready for auth'
7		GetAccessory-AuthenticationInfo	no params
The accessory proceeds with authentication, as illustrated in Table 2-8 (page 106).			
n		SetAccessoryStatus-Notification	send mask value of 0x00000004 to set notifications of accessory fault conditions
n+1	RetAccessoryStatus-Notification		send mask value of 0x00000004 to acknowledge that notifications of accessory fault conditions are set
n+2	AccessoryStatus-Notification		send data values of 0x00,0x00 to initialize fault monitoring in the Apple device to 'no fault' status and mode (see Table 3-108 (page 181)).

3.3.53 Command 0x49: SetEventNotification

Lingo: 0x00 — Origin: Accessory

This command enables asynchronous remote event notifications for specific Apple device events, as described in "Status Notifications From Apple Devices" (page 48).

When the accessory identifies itself, it can check the attached Apple device's notification support by sending a `GetSupportedEventNotification` command. If the Apple device supports notifications, it returns a `RetSupportedEventNotification` command confirming that it supports at least Flow Control notifications (bit 2 in the 64-bit big-endian notification bitmask). If the accessory needs to receive notifications of types that the attached Apple device supports, it must then send this command to register for one or more of the notifications listed in Table 3-112 (page 184).

The Apple device acknowledges this command with a General Lingo `iPodAck` command reporting Status OK (0x00).

If the accessory registers for camera notifications, the Apple device also sends an `iPodNotification` command reporting its current Camera mode. The Apple device may send these commands in any order.

For a suggested sequence of commands to enable Apple device notifications, see "Status Notifications From Apple Devices" (page 48).

Table 3-111 `SetEventNotification` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	8	Event notification mask; see Table 3-112 (page 184).
Packet footer specified in "Command Packets" (page 109).		

Table 3-112 Notification bitmask bits

Bit number	Description	Notes
0-1	Reserved; set to 0	
2	Flow control	Must always set to 1. If and only if the accessory uses IDPS, Flow Control notifications are enabled by default. See "Identification" (page 93).
3	Radio tagging status	See "iTunes Tagging Accessory Design" (page 467).
4	Camera status	See "Accessory Control of the iPod 5G nano Camera" (page 218).
5	Charging info	See "Command 0x54: SetAvailableCurrent" (page 204).
6-8	Reserved; set to 0	
9	Database Changed	See Table 3-121 (page 189).
10	`NowPlayingApplicationBundleName` status	See "Accessory Launching of iOS Applications" (page 50) and "Interaction with iOS Media Applications" (page 54).
11	`SessionSpaceAvailable` status	See Table 3-123 (page 189).
12	Reserved; set to 0	

Bit number	Description	Notes
13	Command completed	See Table 3-124 (page 189). This notification can be used to determine completion of the `CreateGeniusPlaylist` (Lingoes 0x03 and 0x04) and `RefreshGeniusPlaylist` (Lingo 0x04) commands, because they may take up to 30 seconds to finish.
14	Reserved; set to 0	
15	Status of iPod Out Mode	See "iPod Out Mode" (page 78).
16	Reserved; set to 0	
17	Bluetooth connection status	See "Bluetooth Autopairing and Connection Status Notifications" (page 55).
18	Reserved; set to 0	
19	`NowPlaying-ApplicationDisplayName` status	See Table 3-128 (page 191).
20	AssistiveTouch status	See Table 3-129 (page 191).
21-63	Reserved; set to 0	

3.3.54 Command 0x4A: iPodNotification

Lingo: 0x00 — Origin: Apple device

If an accessory has registered for notifications, using `SetEventNotification`, the Apple device sends this command to the accessory every time there is a change in the state of any of the processes for which notification was requested. The notification timings and payloads are different for each of the notification types listed in Table 3-112 (page 184):

- If Flow Control notifications are enabled, the Apple device sends a Flow Control notification whenever the incoming queue is full and the Apple device is unable to accept more packets. When an accessory receives this notification, it must stop sending packets to the Apple device for the wait time specified by the notification packet shown in Table 3-113 (page 187). If transaction IDs are being used, the Flow Control notification also returns the transaction ID of the packet that caused the overflow. This packet was not added to the incoming data queue; hence the accessory can use the overflow transaction ID to determine which packets need to be resent after the wait time.
- If the accessory registers for Radio Tagging notifications, the Apple device sends notifications when tagging information is available or when a tagging operation is initiated (either from the Apple device's user interface or from the accessory via the Simple Remote lingo `RadioButtonStatus` command). The notification packet for Radio Tagging is shown in Table 3-114 (page 187).
- If the accessory registers for Camera notifications, the Apple device sends a notification immediately after receiving `SetEventNotification` and subsequently whenever its Camera mode changes. It passes the state of its Camera mode using the packet format shown in Table 3-116 (page 188). Because the Apple device's notification mechanism is asynchronous, the accessory may receive this packet before its `SetEventNotification` command has been acknowledged. The Apple device may also send

notifications to the accessory at any time as the result of changes in the Camera application's state that may be unrelated to any action taken by the accessory. For full details see "Accessory Control of the iPod 5G nano Camera" (page 218).

- If the accessory registers for charging notifications, the Apple device sends a notification immediately after receiving `SetEventNotification` and subsequently whenever its charging info changes. See "Command 0x54: SetAvailableCurrent" (page 204). The Apple product passes the charging information using the packet format shown in Table 3-118 (page 188).
- If the accessory registers for Database Changed notifications, the Apple device sends a notification every time the contents of its database changes. For details, see "Unexpected Database Changes" (page 63). The notification packet for Database Changed notifications is shown in Table 3-121 (page 189).
- If the accessory registers for `NowPlayingApplicationBundleName` notifications, the Apple device sends a notification when the currently active media playback app changes. The media playback app may be different than the app that manages the user interface. The notification packet for `NowPlayingApplicationBundleName` is shown in Table 3-122 (page 189).
- If the accessory registers for `SessionSpaceAvailable` notifications, the Apple device sends a notification when its data space for `AccessoryDataTransfer` commands transitions from Full to Space Available. If the accessory receives an `iPodAck` status of `SessionWriteFailure` (0x17), it must wait for this notification before retrying an `AccessoryDataTransfer` command in the same session. The accessory should try sending an `AccessoryDataTransfer` command with a small data payload at first, to avoid immediately generating another `SessionWriteFailure` error.

  The accessory can check for the availability of this notification by sending a `GetSupportedEventNotification` command and checking whether bit position 11 is set in the returned `Event Notification Mask`. The accessory can determine whether this notification is set by sending a `GetEventNotification` command and checking the same bit. The notification packet for `SessionSpaceAvailable` is shown in Table 3-123 (page 189).
- If the accessory registers for Command Complete notifications, the Apple device sends an asynchronous notification when it completes the command it is currently executing. An example is when the accessory has sent the Apple device an Extended Interface lingo `CreateGeniusPlaylist` command; the Apple device responds first with an `iPodAck` command to indicate that it is working on the playlist and then an `iPodNotification` Command Complete when it has finished. The notification packet for Command Complete is shown in Table 3-124 (page 189).
- If the accessory registers for iPod Out Mode status notifications, the Apple device sends a notification every time iPod Out Mode becomes active or inactive; see "iPod Out Mode" (page 78). The Apple product passes the status information using the packet format shown in Table 3-125 (page 190).
- If the accessory registers for Bluetooth Connection Status notifications, the Apple device sends a notification any time there is a change in the connection status of any unique Bluetooth MAC address that it owns. For details, see "Bluetooth Autopairing and Connection Status Notifications" (page 55). The notification packet for Bluetooth Connection Status notifications is shown in Table 3-126 (page 190).
- If the accessory registers for `NowPlayingApplicationDisplayName` notifications, the Apple device sends a notification any time there is a change in the name of the application that currently has its playing focus. This is the name that is typically displayed on the Apple device's home screen. The notification packet for `NowPlayingApplicationDisplayName` notifications is shown in Table 3-128 (page 191).
- If the accessory registers for AssistiveTouch Status notifications, the Apple device sends a notification every time its AssistiveTouch feature is turned on or off. See "AssistiveTouch" (page 82). The notification packet for AssistiveTouch Status notifications is shown in Table 3-129 (page 191).

Note: The Apple device may send `iPodNotification` commands at any time, including during the IDPS process. The accessory must not acknowledge these commands and must simply ignore any that it cannot handle. The Apple device does not retry these commands.

Table 3-113 `iPodNotification` packet for Flow Control Notifications

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x02	1	Notification type: Flow Control
0x*NN*	4	Wait time in ms
0x*NN*	2	Overflow transaction ID ; Transaction ID of the packet that caused the Apple device's incoming data queue to overflow. If the Apple device does not support IDPS, this and the next byte are omitted.
Packet footer specified in "Command Packets" (page 109).		

Table 3-114 `iPodNotification` packet for Radio Tagging Notifications

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x03	1	Notification type: Radio Tagging
0x*NN*	1	Tag status; see Table 3-115 (page 187).
Packet footer specified in "Command Packets" (page 109).		

Table 3-115 `iPodNotification` payload for Radio Tagging notifications

Byte value	Description
0x00	Tagging operation successful
0x01	Tagging operation failed
0x02	Information available for tagging (all required RT+ information has been received)
0x03	Information not available for tagging (tagging is not possible because the RT+ information has been reset due to a song change, station change, loss of signal, etc.)
0x04-0xFF	Reserved

Table 3-116 `iPodNotification` packet for Camera notifications

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x04	1	Notification type: Camera Notifications
0x*NN*	1	Payload; see Table 3-117 (page 188).
Packet footer specified in "Command Packets" (page 109).		

Table 3-117 `iPodNotification` payload for Camera Notifications

Byte value	Description
0x00	Camera App Off
0x01-0x02	Reserved
0x03	Preview
0x04	Recording
0x05-0xFF	Reserved

Table 3-118 `iPodNotification` packet for charging notifications

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x05	1	Notification type: Charging Info
0x*NN*	1	Charging Info type: see Table 3-119 (page 188).
0x*NN*	2	Charging Info value : see Table 3-120 (page 189)
Packet footer specified in "Command Packets" (page 109).		

Table 3-119 InfoType values for charging notifications

Value	Meaning
0x00	Available current
0x01-0xFF	Reserved

Table 3-120 Charging Info values for iPodNotification

Value	Bytes	Meaning
0x00	2	Available current: specifies the Apple device's new absolute current-sink limit setting in mA.
0x01-0xFF	Reserved	

Table 3-121 `iPodNotification` packet for Database Changed notifications

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x09	1	Notification type: `DatabaseChanged`
Packet footer specified in "Command Packets" (page 109).		

Table 3-122 `iPodNotification` packet for NowPlayingApplicationBundleName notifications

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x0A	1	Notification type: `NowPlayingApplicationBundleName`
0x*NN*	*NN*	Null-terminated UTF-8 string containing the Application ID of the application that has just now received playing focus. Apple-owned media playback apps that can communicate using iAP pass the string 'com.apple.mobileipod'.
Packet footer specified in "Command Packets" (page 109).		

Table 3-123 `iPodNotification` packet for Session Space Available notifications

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x0B	1	Notification type: `SessionSpaceAvailable`
0x*NN*	2	`SessionID`: ID of session with data space now available
Packet footer specified in "Command Packets" (page 109).		

Table 3-124 `iPodNotification` packet for Command Complete notifications

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x0D	1	Notification type: Command Complete

Value	Bytes	Parameter
0x*NN*	1	`LingoID`: Lingo ID of the command that has completed
0x*NN*	2	`CmdID`: ID of the command that has completed
0x*NN*	1	Command completion status: 0x00: Command successful 0x01: Command failed 0x02: Command cancelled 0x03-0xFF: Reserved
Packet footer specified in "Command Packets" (page 109).		

Table 3-125 `iPodNotification` packet for iPod Out Mode status notifications

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x0F	1	Notification type: iPod Out Mode status
0x*NN*	1	Status: 1 = iPod Out Mode active, 0 = iPod Out Mode inactive.
Packet footer specified in "Command Packets" (page 109).		

Table 3-126 `iPodNotification` packet for Bluetooth Connection Status notifications

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x11	1	Notification type: Bluetooth Connection Status
0x*NN*	6	The accessory's Bluetooth MAC address for this Connection Status notification
0x*NN*	8	Each bit indicates the current state of the accessory's Bluetooth connection to the Apple device (via the MAC address) for one of the profiles listed in Table 3-127 (page 190): 1 = connected, 0 = not connected.
Packet footer specified in "Command Packets" (page 109).		

Table 3-127 Profile ID bits for Bluetooth Connection Status notifications

Bit	Meaning
00	Hands Free Profile (HFP)
01	Phone Book Access Profile (PBAP)
02	Reserved

Bit	Meaning
03	Audio/Video Remote Control Profile (AVRCP)
04	Advanced Audio Distribution Profile (A2DP)
05	Human Interface Device Profile (HID)
06	Reserved
07	Apple iAP Profile
08	Personal Area Network–Network Access Point Profile (PAN-NAP or Network Sharing)
09-11	Reserved
12	Personal Area Network–User Profile (PAN-U or Network Consumer)
13-63	Reserved

Table 3-128 `iPodNotification` packet for NowPlayingApplicationDisplayName notifications

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x13	1	Notification type: `NowPlayingApplicationDisplayName`
0x*NN*	*NN*	Null-terminated UTF-8 string containing the full name of the application that has just now received playing focus. This string is typically displayed on the Apple device's home screen.
Packet footer specified in "Command Packets" (page 109).		

Table 3-129 `iPodNotification` packet for AssistiveTouch notifications

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x14	1	Notification type: Assistive Touch status
0x*NN*	1	0x00 = Off, nonzero = On. Notifications are sent at the time of registration and any time AssistiveTouch is turned on or off.
Packet footer specified in "Command Packets" (page 109).		

3.3.55 Command 0x4B: GetiPodOptionsForLingo

Lingo: 0x00 — Origin: Accessory

The accessory sends this command to ask the Apple device to return a 64-bit field that defines the options that the Apple device supports for a specific lingo, identified in `LingoID`. In response, the Apple device sends a `RetiPodOptionsForLingo` command.

If the accessory requests options for the General lingo (0x00) and the Apple device returns an `iPodAck` with nonzero status, then the Apple device does not support `GetiPodOptionsForLingo`; the accessory should use `RequestLingoProtocolVersion` instead. If the accessory requests options for any other lingo and the Apple device returns a nonzero `iPodAck`, that lingo is not listed in Table 3-132 (page 192) or is not supported by the Apple device on the port being used; no `RetiPodOptionsForLingo` command will be returned.

Table 3-130 `GetiPodOptionsForLingo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`LingoID`: ID of lingo for which options are requested; see Table 3-132 (page 192).
Packet footer specified in "Command Packets" (page 109).		

3.3.56 Command 0x4C: RetiPodOptionsForLingo

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command in response to a `GetiPodOptionsForLingo` command from the accessory. It returns the ID of the lingo for which options were requested in `LingoID` and the option bits as a 64-bit big-endian field in bytes 6-13. Table 3-132 (page 192) lists the available option bit values for various lingoes, and Table 3-133 (page 196) lists a sample command flow that queries an Apple device's options for those lingoes.

Table 3-131 `RetiPodOptionsForLingo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`LingoID`: ID of lingo for which options were requested.
0x*NN*	8	Option bits
Packet footer specified in "Command Packets" (page 109).		

Table 3-132 `RetiPodOptionsForLingo` option bits

Lingo	Lingo ID	Bit	Description
General	0x00	00	Line out usage
		01	Video output
		02	NTSC video signal format

Lingo	Lingo ID	Bit	Description
		03	PAL video signal format
		04	Composite video out connection
		05	S-Video video out connection
		06	Component video out connection
		07	Closed Captioning (video)
		08	Video aspect ratio 4:3 (fullscreen)
		09	Video aspect ratio 16:9 (widescreen)
		10	Subtitles (video)
		11	Video Alternate Audio Channel
		12	Reserved
		13	Communication with iOS applications
		14	Apple device notifications
		15-18	Reserved
		19	Pause on power removal preference control
		20-22	Reserved
		23	0 = restricted IDPS token handling. If this bit is 0, the Apple device's IDPS process can handle only `IdentifyToken`, `AccessoryCapsToken`, `AccessoryInfoToken`, `iPodPreferenceToken`, `EAProtocolToken`, and `BundleSeedIDPrefToken`. See Table 3-67 (page 161).
		24	Request Application Launch; see "Accessory Launching of iOS Applications" (page 50).
		25	Reserved
		26	1 = The Apple device supports switching between Apple device analog line out and USB Device mode audio output routings using the General lingo `SetiPodPreference` line out preference. This must be used by USB host accessories with Apple devices that don't support the Digital Audio lingo `SetVideoDelay` command for maintaining audio/video stream synchronization when video media is playing See "Audio/Video Synchronization" (page 347). 0 = The Apple device does not support switching audio routings using the `SetiPodPreference` command. To switch between USB Device mode audio and analog line out audio routings, the accessory must identify and authenticate itself repeatedly with and without the Digital Audio lingo.

Lingo	Lingo ID	Bit	Description
		27	USB Host Mode invoked by hardware, using accessory ID resistor; see *MFi Accessory Hardware Specification,* Release R4 or later.
		28	USB Host Mode supports audio output streaming to USB audio devices; valid only if bit 27 is set.
		29	USB Host Mode supports audio input streaming from USB audio devices; valid only if bit 27 is set.
		30	USB Host Mode does not support maximum current draw in Low Power Mode; see "Low Power Mode" in *MFi Accessory Hardware Specification*, release R7 or later.
		31-63	Reserved
Simple Remote	0x02	00	Context-specific controls
		01	Audio media controls
		02	Video media controls
		03	Image media controls
		04	Sports media controls
		05-07	Reserved
		08	Camera media controls
		09	USB HID commands
		10	VoiceOver controls
		11	VoiceOver preferences
		12	AssistiveTouch cursor support; see "AssistiveTouch" (page 82).
		13-63	Reserved
Display Remote	0x03	00	UI Volume control
		01	Absolute Volume control
		02	Supports Genius Playlist creation
		03-63	Reserved
Extended Interface	0x04	00	Video browsing

Lingo	Lingo ID	Bit	Description

		01	The following Extended Interface commands are enabled:
			`GetDBiTunesInfo`
			`RetDBiTunesInfo`
			`GetUIDTrackInfo`
			`RetUIDTrackInfo`
			`GetDBTrackInfo`
			`RetDBTrackInfo`
			`GetPBTrackInfo`
			`RetPBTrackInfo`

		02	Nested playlists
		03	Supports Genius Playlist creation and refresh
		04	Apple device displays images sent by Extended Interface command "Command 0x0032: SetDisplayImage" (page 435).
		05	The accessory can access the list of categories that the he Apple device supports
		06	Supports `PlayControl` Play and Pause commands
		07	Supports UID-based commands; see "Command 0x004C: GetArtworkTimes" (page 458) through "Command 0x004F: RetArtworkData" (page 460).
		08-63	Reserved
Accessory Power	0x05	00-63	Reserved
USB Host Mode	0x06	00	**Deprecated;** use General lingo option bit 27 to indicate that USB Host Mode is invoked by hardware.
		01	USB Host Mode invoked by firmware; see "USB Host Mode" (page 88).
		02-63	Reserved
RF Tuner	0x07	00	RDS Raw Mode support
		01	HD Radio Tuning support
		02	AM Radio Tuning support
		03-63	Reserved
Accessory Equalizer	0x08	00-63	Reserved
Sports	0x09	00	Reserved

Lingo	Lingo ID	Bit	Description
		01	Nike + iPod Cardio Equipment
		02-63	Reserved
Digital Audio	0x0A	00	A/V Synchronization: 1=`SetVideoDelay` enabled, 0=`SetVideoDelay` disabled. See "Audio/Video Synchronization" (page 347), Step 3.
		01	Reserved
		02	Supports accessory reporting of sample rate capabilities using a FID token; see "USB Device Mode Audio" (page 347).
		03	Supports accessory reporting of video synchronization delay using a FID token; see "Audio/Video Synchronization" (page 347), Step 5.
		04-63	Reserved
Storage	0x0C	00	iTunes Tagging
		01	Nike + iPod Cardio Equipment
		02-63	Reserved
iPod Out	0x0D	00	iPod Out wheel UI mode is available
		01	Reserved
		02	PAL video enabled
		03-63	Reserved
Location	0x0E	00	The Apple device accepts NMEA GPS location data
		01	The Apple device can send location assistance data
		02-63	Reserved

Table 3-133 (page 196) lists a sample command flow that queries an Apple device's options for most of the iAP lingoes.

Table 3-133 Sample GetiPodOptionsForLingo and RetiPodOptionsForLingo commands

Step	Accessory command	Apple device command	Comment
1	`StartIDPS`		no params
2		`iPodAck`	acknowledging 'Success (OK)' to command 'General Lingo::StartIDPS'

Step	Accessory command	Apple device command	Comment
If the `iPodAck` reply to `StartIDPS` returns a status of 0x00, the accessory proceeds to Step 3. Otherwise, ■ If the accessory receives no `iPodAck` command within 500 ms, it may retry Step 1 at 500 ms intervals as long as Apple Device Detect (pin 30 of the 30-pin connector) is low and Accessory Power (pin 13) is high. Accessories that don't monitor these signals (such as Bluetooth accessories or USB accessories without a 191 kΩ ID resistor) may retry without checking them. ■ If the accessory receives an `iPodAck` status of 0x04 (Bad Parameter), the iPod does not support IDPS. The accessory must not proceed to Step 3. Instead, it must perform either one of these two alternative actions: ❑ If IDPS support is required, within 800 ms the accessory must send an `IdentifyDeviceLingoes` command with all fields set to 0xFF. This causes the iPod to display an "Accessory not supported" message. ❑ If IDPS support is not required, the accessory must then send another `IdentifyDeviceLingoes` command with its lingo mask set for only the General lingo, with no options, and with a Device ID of 0x00. See "Canceling a Current Authentication Process With IdentifyDeviceLingoes" (page 135). A sample command sequence is listed in Table C-48 (page 553). ■ If the accessory receives any other nonzero `iPodAck` status, it must not send any more iAP commands until it has been disconnected. Upon reconnection it must restart identification at Step 1.			
3	`GetiPodOptions-ForLingo`		getting options for General Lingo
4		`RetiPodOptions-ForLingo`	returning options of 0000000000006FFF (Line out usage | Video output | NTSC video signal format | PAL video signal format | Composite video out connection | S-Video video out connection | Component video out connection | Closed captioning (video) | Video aspect ratio 4:3 (fullscreen) | Video aspect ratio 16:9 (widescreen) | Subtitles (video) | Video Alternate Audio Channel | App communication capable | iPod notifications) for General Lingo
5	`GetiPodOptions-ForLingo`		getting options for Simple Remote Lingo
6		`RetiPodOptions-ForLingo`	returning options of 000000000000000F (Audio media controls | Video media controls | Image media controls | context-specific controls) for Simple Remote Lingo
7	`GetiPodOptions-ForLingo`		getting options for Display Remote Lingo

Step	Accessory command	Apple device command	Comment
8		RetiPodOptions-ForLingo	returning options of 0000000000000001 (Volume control) for Display Remote Lingo
9	GetiPodOptions-ForLingo		getting options for Extended Interface Lingo
10		RetiPodOptions-ForLingo	returning options of 0000000000000001 (Video browsing) for Extended Interface Lingo
11	GetiPodOptions-ForLingo		getting options for Accessory Equalizer Lingo
12		RetiPodOptions-ForLingo	returning options of 0000000000000000 (none) for Accessory Equalizer Lingo
13	GetiPodOptions-ForLingo		getting options for Sports Lingo
14		RetiPodOptions-ForLingo	returning options of 0000000000000002 (Nike + iPod cardio equipment) for Sports Lingo
15	GetiPodOptions-ForLingo		getting options for Digital Audio Lingo
16		RetiPodOptions-ForLingo	returning options of 0000000000000000 (none) for Digital Audio Lingo
17	GetiPodOptions-ForLingo		getting options for Storage Lingo
18		RetiPodOptions-ForLingo	returning options of 0000000000000003 (iTunes tagging | Nike + iPod cardio equipment) for Storage Lingo
19	GetiPodOptions-ForLingo		getting options for Location Lingo
20		RetiPodOptions-ForLingo	returning options of 0000000000000003 (Apple device accepts NMEA GPS location data | Apple device can send location assistance data) for Location Lingo
21	SetFIDTokenValues		setting 1 FID tokens ; IdentifyToken = (lingoes: 0/2 | options 0x00000002 | device ID 0x00000200)
22		AckFIDTokenValues	1 ACKs for FID tokens ; IdentifyToken = accepted

Step	Accessory command	Apple device command	Comment
23	SetFIDTokenValues		setting 8 FID tokens ; AccInfoToken = Acc name (Apple device Accessory Name); AccInfoToken = Acc FW version (v1.2.1); AccInfoToken = Acc HW version (v1.0.0); AccInfoToken = Acc manufacturer (Apple Inc.); AccInfoToken = Acc model number (MA1390LL/A); EAProtocolToken = 1 (com.apple.protocolMain); EAProtocolToken = 2 (com.apple.protocolAlternative); BundleSeedIDString = 24D4XFAF43
24		AckFIDTokenValues	8 ACKs for FID tokens ; AccInfoToken = (Acc name) accepted; AccInfoToken = (Acc FW version) accepted; AccInfoToken = (Acc HW version) accepted; AccInfoToken = (Acc manufacturer) accepted; AccInfoToken = (Acc model number) accepted; EAProtocolToken = (1) accepted; EAProtocolToken = (2) accepted; BundleSeedIDPrefToken = accepted
25	SetFIDTokenValues		setting 1 FID tokens ; AccCapsToken = 0x0000000000000A17
26		AckFIDTokenValues	1 ACKs for FID tokens ; AccCapsToken = accepted
27	SetFIDTokenValues		setting 1 FID tokens ; iPodPreferenceToken = (setting 'used' for preference class 'line out usage' with restore on exit 'selected')
28		AckFIDTokenValues	1 ACKs for FID tokens ; iPodPreferenceToken = (line out usage) accepted
29	SetFIDTokenValues		setting 1 FID tokens ; iPodPreferenceToken = (setting 'on' for preference class 'video out setting' with restore on exit 'selected')
30		AckFIDTokenValues	1 ACKs for FID tokens ; iPodPreferenceToken = (video out setting) accepted

Step	Accessory command	Apple device command	Comment
31	SetFIDTokenValues		setting 1 FID tokens ; iPodPreferenceToken = (setting 'widescreen' for preference class 'screen configuration' with restore on exit 'selected')
32		AckFIDTokenValues	1 ACKs for FID tokens ; iPodPreferenceToken = (screen configuration) accepted
33	SetFIDTokenValues		setting 1 FID tokens ; iPodPreferenceToken = (setting 'PAL' for preference class 'video format setting' with restore on exit 'selected')
34		AckFIDTokenValues	1 ACKs for FID tokens ; iPodPreferenceToken = (video format setting) accepted
35	SetFIDTokenValues		setting 1 FID tokens ; iPodPreferenceToken = (setting 'component' for preference class 'video connection' with restore on exit 'selected')
36		AckFIDTokenValues	1 ACKs for FID tokens ; iPodPreferenceToken = (video connection) accepted
37	SetFIDTokenValues		setting 1 FID tokens ; iPodPreferenceToken = (setting '16:9' for preference class 'aspect ratio' with restore on exit 'selected')
38		AckFIDTokenValues	1 ACKs for FID tokens ; iPodPreferenceToken = (aspect ratio) accepted
39	SetFIDTokenValues		setting 1 FID tokens ; iPodPreferenceToken = (setting 'component' for preference class 'video connection' with restore on exit 'selected')
40		AckFIDTokenValues	1 ACKs for FID tokens ; iPodPreferenceToken = (video connection) accepted
41	SetFIDTokenValues		setting 1 FID tokens ; iPodPreferenceToken = (setting 'on' for preference class 'video subtitles' with restore on exit 'selected')

Step	Accessory command	Apple device command	Comment
42		AckFIDTokenValues	1 ACKs for FID tokens ; iPodPreferenceToken = (video subtitles) accepted
43	SetFIDTokenValues		setting 1 FID tokens ; iPodPreferenceToken = (setting 'on' for preference class 'alternate audio channel' with restore on exit 'selected')
44		AckFIDTokenValues	1 ACKs for FID tokens ; iPodPreferenceToken = (alternate audio channel) accepted
45	SetFIDTokenValues		setting 1 FID tokens ; iPodPreferenceToken = (setting 'on' for preference class 'closed captions' with restore on exit 'selected')
46		AckFIDTokenValues	1 ACKs for FID tokens ; iPodPreferenceToken = (closed captions) accepted
47	EndIDPS		status 'finished with IDPS; proceed to authentication'
48		IDPSStatus	status 'ready for auth'
49		GetAccessory-AuthenticationInfo	no params
50	RetAccessory-AuthenticationInfo		returning auth protocol v2.0; section: 0/1; cert data: ...
51		iPodAck	acknowledging 'Success (OK)' to command 'General Lingo::RetAccessoryAuthenticationInfo'
52	RetAccessory-AuthenticationInfo		returning auth protocol v2.0; section: 1/1; cert data: ...
53		AckAccessory-AuthenticationInfo	acknowledging 'auth info supported'
54		GetAccessory-Authentication-Signature	offering challenge '...' with retry counter 1
55	RetAccessory-Authentication-Signature		returning signature '...'

Step	Accessory command	Apple device command	Comment
56		AckAccessory-AuthenticationStatus	acknowledging authentication status 'Success (OK)'

3.3.57 Command 0x4D: GetEventNotification

Lingo: 0x00 — Origin: Accessory

The accessory sends this command to the Apple device to ask for the Apple device's current 64-bit big-endian notification bitmask.

Table 3-134 GetEventNotification packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

3.3.58 Command 0x4E: RetEventNotification

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command to the accessory in response to a GetEventNotification command. It passes its current 64-bit big-endian notification bitmask.

Table 3-135 RetEventNotification packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x00	8	Event notification mask; see Table 3-112 (page 184).
Packet footer specified in "Command Packets" (page 109).		

3.3.59 Command 0x4F: GetSupportedEventNotification

Lingo: 0x00 — Origin: Accessory

The accessory sends this command to the Apple device to query the Apple device's support for notifications. The Apple device responds with a RetSupportedEventNotification command.

Table 3-136 `GetSupportedEventNotification` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

3.3.60 Command 0x50: CancelCommand

Lingo: 0x00 — Origin: Accessory

This command lets an accessory cancel a multipacket data transfer request that it has sent to an Apple device. It requires that the accessory use unique transaction IDs for its commands and that it has declared its ability to handle multipacket downloads by setting bit 19 in the IDPS Accessory capabilities token; see Table 3-71 (page 164). This command may be used to cancel only the following commands:

Simple Remote lingo `GetCurrentItemProperty` command
Display Remote lingo `GetTrackArtworkData` command
Extended Interface lingo `GetTrackArtworkData` command
Extended Interface lingo `RetrieveCategorizedDatabaseRecords` command

If the `CancelCommand` is successful, the Apple device responds with an `iPodAck` command passing 0x00 and the `CancelCommand`'s transaction ID. A typical use for `CancelCommand` is to halt a long download of artwork from the Apple device if the user switches to a different track.

Table 3-137 `CancelCommand` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`LingoID`: ID of lingo of command being cancelled
0x*NN*	2	`CommandID`: Command ID of command being cancelled.
0x*NN*	2	`TransactionID`: Transaction ID of command being cancelled.
Packet footer specified in "Command Packets" (page 109).		

3.3.61 Command 0x51: RetSupportedEventNotification

Lingo: 0x00 — Origin: Apple device

The Apple device sends this command to the accessory in response to a `GetSupportedEventNotification` command. It passes a 64-bit big-endian bitmask that defines the types of notifications that the Apple device supports.

Table 3-138 `RetSupportedEventNotification` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x00	8	Event notification mask; see Table 3-112 (page 184).
Packet footer specified in "Command Packets" (page 109).		

3.3.62 Command 0x54: SetAvailableCurrent

Lingo: 0x00 — Origin: Accessory

If an accessory is designed to provide power to an Apple device, it sends this command to set the absolute current-sink limit that the Apple device may draw, as specified in "Power Management" (page 48).

If 5 V is present on USB V_{BUS} (pin 8 of the 30-pin connector), and if the limit set by this command is higher than the existing limit, the Apple device may immediately switch to the new limit.

Whether the limit is raised or lowered, the Apple device will send an `iPodNotification` command after it has updated its current-sink limit value. To receive this notification, the accessory must have previously registered for Charging Info notifications by sending a General lingo `SetEventNotification` command that sets bit 5, as listed in Table 3-112 (page 184).

Table 3-139 `SetAvailableCurrent` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	2	`CurrentLimit`: Maximum current in mA that the Apple device may draw.
Packet footer specified in "Command Packets" (page 109).		

3.3.63 Command 0x56: SetInternalBatteryChargingState

Lingo: 0x00 — Origin: Accessory

An accessory can use this command to request that an attached Apple device charge or not charge its internal battery. The default state of the Apple device is to charge its internal battery from power supplied by the accessory; otherwise, it tries to operate on that power. When the Apple device is detached it reverts to its default state.

Table 3-140 `SetInternalBatteryChargingState` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		

Value	Bytes	Parameter
0x*NN*	1	`ChargingState` requested; see Table 3-141 (page 205).
Packet footer specified in "Command Packets" (page 109).		

Table 3-141 Charging state values

Value	Description
0x00	The Apple device should not charge its internal battery.
0x01	The Apple device may charge its internal battery.
0x02-0xFF	Reserved

3.3.64 Command 0x64: RequestApplicationLaunch

Lingo: 0x00 — Origin: Accessory

The accessory sends this command to an Apple device to request that it launch a specific application. The accessory passes an Application Bundle ID string, such as `"com.mycompany.myapp"`, to specify which application to launch. The Apple device replies with an `iPodAck` command that returns the application launch status.

Note: an `iPodAck` status of OK (0x00) only indicates that the launch is possible; it does not guarantee that the application is running at that time. A status of Command Failed (0x02) indicates that the application either does not exist on the Apple device or that the Apple device is in a condition that prevents the launch. In this case, the Accessory must not retry the `RequestApplicationLaunch` command.

Table 3-142 `RequestApplicationLaunch` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x00	1	Reserved
0x02	1	Set to 0x02. If the Apple device returns an `iPodAck` command with a Bad Parameter status (0x04), repeat the `RequestApplicationLaunch` command with this byte set to 0x01.
0x00	1	Reserved
0x*NN*	*NN*	Null-terminated UTF-8 string containing the Application ID of the application to be launched.
Packet footer specified in "Command Packets" (page 109).		

3.3.65 Command 0x65: GetNowPlayingApplicationBundleName

Lingo: 0x00 — Origin: Accessory

An accessory sends this command to an Apple device to determine which application has the current Now Playing focus (the actual playing of media, which may be different from the user interface focus). The Apple device responds with a `RetNowPlayingApplicationBundleName` command.

Table 3-143 `GetNowPlayingApplicationBundleName` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

3.3.66 Command 0x66: RetNowPlayingApplicationBundleName

Lingo: 0x00 — Origin: Apple device

An Apple device sends this command to an accessory, in response to a `GetNowPlayingApplicationBundleName` command, to report which application has the current Now Playing focus.

Table 3-144 `RetNowPlayingApplicationBundleName` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	*NN*	Null-terminated UTF-8 string containing the Application ID of the application that has the current Now Playing focus.
Packet footer specified in "Command Packets" (page 109).		

3.3.67 Command 0x67: GetLocalizationInfo

Lingo: 0x00 — Origin: Accessory

An accessory sends this command to its attached Apple device to determine the device's localization. The Apple device responds with a `RetLocalizationInfo` command. Depending on the value it passes in byte 7, the accessory can request information about the Apple device's language or regional localization.

Table 3-145 `GetLocalizationInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		

Value	Bytes	Parameter
0x*NN*	1	`type`: localization info type; 0x00 = language, 0x01 = region. Other values are reserved.
Packet footer specified in "Command Packets" (page 109).		

3.3.68 Command 0x68: RetLocalizationInfo

Lingo: 0x00 — Origin: Apple device

An Apple device replies to `GetLocalizationInfo` by sending this command. It returns the information type requested and a null-terminated UTF-8 string of localization tags that conforms to the IETF Best Current Practices (BCP) 47 specification. Strings of this type are also used by Apple's Cocoa Core Foundation classes. If any unsupported or reserved type of localization information is requested, the command returns a General lingo `iPodAck` command with an error code of 0x04 (Bad Parameter).

When reporting language, the command returns the the ISO 639 language code (such as "en-GB" for British English) for the language currently selected on the Apple device in Settings > General > International > Language. This response may be limited to the language tag, without script, region or variant tags, if more detailed information is not available.

When reporting regional information the command returns the region code for the language currently selected on the Apple device in Settings > General > International > Region, such as en_GB for the UK. Note the use of an underscore rather than a hyphen to punctuate region tags. The region tag may define the same users as the language tag.

Table 3-146 `RetLocalizationInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`type`: info type requested by `GetLocalizationInfo`.
0x*NN*	*NN*	Null-terminated UTF-8 string conforming to the IETF BCP 47 specification.
Packet footer specified in "Command Packets" (page 109).		

3.3.69 Command 0x69: RequestWiFiConnectionInfo

Lingo: 0x00 — Origin: Accessory

An accessory sends this command to its attached Apple device to request the device's current Wi-Fi connection information. The Apple device replies with a `WiFiConnectionInfo` command if and only if the `RequestWiFiConnectionInfo` command has been successfully acknowledged. The Accessory must not retry this command.

Table 3-147 `RequestWiFiConnectionInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

3.3.70 Command 0x6A: WiFiConnectionInfo

Lingo: 0x00 — Origin: Apple device

An Apple device replies to `RequestWiFiConnectionInfo` by sending this command if and only if it has successfully acknowledged the `RequestWiFiConnectionInfo` command. It sends Wi-Fi connection data only if the value of byte 5 (Status) is 0x00, and if the value of byte 6 is 0x00 (Unsecured), it does not include the Wi-Fi passphrase. This command is sent asynchronously, so the accessory may receive it at any time.

Table 3-148 `WiFiConnectionInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Status; see Table 3-149 (page 208).
0x*NN*	1	Security type; see Table 3-150 (page 209).
0x*NN*	*NN*	32 octets of `WiFiSSID` information.
0x*NN*	*NN*	Null-terminated UTF-8 string containing `WiFiPassphrase` information.
Packet footer specified in "Command Packets" (page 109).		

Table 3-149 Wi-Fi status values

Value	Description
0x00	Success
0x01	Connection information unavailable
0x02	User declined
0x03	Command failed
0x04-0xFF	Reserved

Table 3-150 Wi-Fi security type values

Value	Description
0x00	Unsecured
0x01	WEP
0x02	WPA
0x03	WPA2
0x04	Mixed WPA and WPA2
0x05-0xFF	Reserved

4. Additional Lingoes

Besides the General lingo, the iPod Accessory Protocol (iAP) defines a number of additional lingoes that accessories can use. This chapter describes these lingoes and their commands.

Table 4-1 (page 211) shows the available accessory lingoes and their IDs.

Table 4-1 Additional iAP lingoes

Lingo	ID	Notes
Microphone	0x01	**Deprecated;** see "Deprecated Lingo 0x01: Microphone Lingo" (page 532).
Simple Remote	0x02	
Display Remote	0x03	
Extended Interface	0x04	See "Extended Interface Mode" (page 397). For a sample command sequence that declares the Extended Interface lingo, see Table 4-293 (page 389).
Accessory Power	0x05	**Deprecated;** see "Deprecated Lingo 0x05: Accessory Power Lingo" (page 547).
USB Host Mode	0x06	
RF Tuner	0x07	
Accessory Equalizer	0x08	
Sports	0x09	
Digital Audio	0x0A	
Reserved	0x0B	
Storage	0x0C	
iPod Out	0x0D	
Location	0x0E	
Reserved	0x0F–0xFF	

4.1 Command Timings

Some iAP commands sent by Apple devices take a significant time to execute, and some of these retry if the accessory does not acknowledge the first attempt. Timeout information and the number of times the command tries to execute are shown in Table 4-2 (page 212).

Table 4-2 Select Apple device command timings

Lingo	Command	Timeout	Tries
RF Tuner (0x07) (See note below)	`GetCaps` (0x01)	200 ms	1
	`GetTunerCtrl` (0x03)	200 ms	1
	`SetTunerCtrl` (0x05)	200 ms	1
	`GetTunerBand` (0x06)	200 ms	1
	`SetTunerBand` (0x08)	200 ms	1
	`GetTunerFreq` (0x09)	200 ms	1
	`SetTunerFreq` (0x0B) (AM mode)	300 ms	1
	`SetTunerFreq` (0x0B) (FM mode)	200 ms	1
	`GetTunerMode` (0x0C)	200 ms	1
	`SetTunerMode` (0x0E)	200 ms	1
	`GetTunerSeekRssi` (0x0F)	200 ms	1
	`SetTunerSeekRssi` (0x11)	200 ms	1
	`TunerSeekStart` (0x12)	200 ms	1
	`GetTunerStatus` (0x14)	200 ms	1
	`GetStatusNotifyMask` (0x16)	200 ms	1
	`SetStatusNotifyMask` (0x18)	200 ms	1
	`GetRdsReadyStatus` (0x1A)	200 ms	1
	`GetRdsData` (0x1C)	200 ms	1
	`GetRdsNotifyMask` (0x1E)	200 ms	1
	`SetRdsNotifyMask` (0x20)	200 ms	1
Accessory Equalizer (0x08)	`GetCurrentEQIndex` (0x01)	2500 ms	3
	`SetCurrentEQIndex` (0x03)	3000 ms	3

Lingo	Command	Timeout	Tries
	GetEQSettingCount (0x04)	3000 ms	3
	GetEQIndexName (0x06)	3000 ms	3
Sports (0x09)	GetAccessoryCaps (0x03)	500 ms	2
Digital Audio (0x0A)	GetAccessorySampleRateCaps (0x02)	3000 ms	3
Location (0x0E)	GetAccessoryCaps (0x01)	500 ms	1
	GetAccessoryControl (0x03)	500 ms	1
	SetAccessoryControl (0x05)	500 ms	1
	GetAccessoryData (0x06)	500 ms	1
	SetAccessoryData (0x08)	500 ms	1

RF Tuner lingo note: The accessory must not exceed the RF Tuner command timings listed when responding to commands from an Apple device.

4.2 Lingo 0x02: Simple Remote Lingo

This lingo is intended for a remote accessory that retains no state information about the Apple device. Simple commands are sent to the Apple device, and no acknowledgment or state information is sent back to the accessory. This lingo is used by the Apple device's standard in-line remote control. For a sample command sequence that declares this lingo, see Table 4-294 (page 391).

4.2.1 History and Applicability

System software versions 2.0 through 2.2 on the 3G iPod and versions 1.0 and 1.1 of the iPod mini support only the first five button responses (0 through 4) in the ContextButtonStatus command. This is called the contextual button lingo and includes only command 0x00. An extended set of commands (0x01 through 0x04) is available in the dedicated media lingoes, as shown in Table 4-3 (page 213).

Table 4-3 Simple remote lingo command summary

Cmd ID	Command name	Direction	Parameters:bytes	Lingo version	Authentication
0x00	ContextButtonStatus	Acc to Dev	{buttonStatus:4}	All	No
0x01	iPodAck	Dev to Acc	{cmdStatus:1, cmdID:1}	1.01	Yes

Cmd ID	Command name	Direction	Parameters:bytes	Lingo version	Authentication
0x02	ImageButtonStatus	**Deprecated;** see "Deprecated Simple Remote Lingo Command" (page 542).			
0x03	VideoButtonStatus	Acc to Dev	{buttonStatus:4}	1.01	Yes
0x04	AudioButtonStatus	Acc to Dev	{buttonStatus:4}	1.01	Yes
0x05–0x0A	Reserved	N/A	N/A	N/A	N/A
0x0B	iPodOutButtonStatus	Acc to Dev	{buttonSource:1, iPodOutButtonMask:<var>}	1.03	Yes
0x0C	RotationInputStatus	Acc to Dev	{rotationSource:1, controllerType:1, rotationDirection:1, rotationAction:1} or {rotationType:1, detentsOrDegreesMoved:2, detentsOrDegreesTotal:2}	1.03	Yes
0x0D	RadioButtonStatus	Acc to Dev	{buttonStatus:1}	1.02	Yes
0x0E	CameraButtonStatus	Acc to Dev	{buttonStatus:1}	1.02	Yes
0x0F	RegisterDescriptor	Acc to Dev	{index:1, VID:2, PID:2, country_code:1, descriptor:<var>}	1.04	Yes
0x10	iPodHIDReport	Acc to Dev	{index:1, report_type:1, report:<var>}	1.04	Yes
0x11	AccessoryHIDReport	Dev to Acc	{index:1, report_type:1, report:<var>}	1.04	Yes
0x12	UnregisterDescriptor	Acc to Dev	{index:1}	1.04	Yes
0x13	VoiceOverEvent	Acc to Dev	{eventType:1, eventData:<var>}	1.04	Yes
0x14	GetVoiceOver-Parameter	Acc to Dev	{paramType:1}	1.04	Yes
0x15	RetVoiceOver-Parameter	Dev to Acc	{paramType:1, paramValue:<var>}	1.04	Yes

Cmd ID	Command name	Direction	Parameters:bytes	Lingo version	Authentication
0x16	SetVoiceOver-Parameter	Acc to Dev	{paramType:1, paramValue:<var>}	1.04	Yes
0x17	GetCurrentVoiceOver-ItemProperty	Acc to Dev	{propertyType:1}	1.04	Yes
0x18	RetCurrentVoiceOver-ItemProperty	Dev to Acc	{propertyType:1, propertyValue:<var>}	1.04	Yes
0x19	SetVoiceOverContext	Acc to Dev	{context:1}	1.04	Yes
0x1A–0x80	Reserved	N/A	N/A	N/A	N/A
0x81	AccessoryAck	Acc to Dev	{cmdStatus:1, cmdIDOrig:1}	1.04	Yes
0x82–0xFF	Reserved	N/A	N/A	N/A	N/A

Dedicated media lingoes are supported by version 1.01 of the Simple Remote lingo. By using GetiPodOptionsForLingo (or querying the version number, as shown in Table 4-4 (page 215) if the Apple device does not support GetiPodOptionsForLingo), an accessory can determine the level of command support.

Table 4-4 Simple remote lingo support versions

Version	Support
No version	Command 0 supported, buttons 0–4
No version	Command 0 supported, buttons 0–25
1.00	Version number can be obtained
1.01	Commands 0–4 supported, buttons 0–25, media controls
1.02	Bug fix for compatibility with some accessories: when an accessory identifies itself using the General lingo Identify command, the 200 ms time limit during which ContextButtonStatus must report that all buttons are up is removed.
1.03	iPod Out support.

Table 4-5 (page 215) shows the history of command changes in the Simple Remote lingo:

Table 4-5 Simple Remote lingo command history

Lingo version	Command changes	Features
No version	Add: 0x00	Context specific button status, buttons 0–4

Lingo version	Command changes	Features
No version	None	Context-specific button status, buttons 5-25 added (4G iPod with firmware versions 3.0.0 and 3.0.1)
1.00	None	Version number available through `RequestLingoProtocol-Version`
1.01	Add: 0x01–0x04	Image-, video-, and audio-specific media control button status
1.02	Add: 0x0D, 0x0E	All-buttons-up timeout applied to all commands, except not to `ContextButtonStatus` when that command is used with the General lingo (0x00) `Identify` command (0x01).
1.04	Add: 0x0B, 0x0C, 0x0F–0x19, 0x81	Support USB Human Interface Accessory (HID) descriptors, support iPod Out, support user interface VoiceOver commands

4.2.2 Playback Engine Playlists

An Apple device's **playback engine** contains a list of queued and currently playing media tracks. Its contents can be queried via the Display Remote or Extended Interface lingoes to obtain track information such as track name, artist, album, genre, etc. One way the playback engine can be loaded is for the user to traverse the Apple device UI, selecting one or more media menu items (such as music or videos) and pressing the play/pause button. Another way is for an accessory to send Extended Interface lingo commands to select database categories and initiate the playback of specific selections.

The list of media tracks currently queued for playback in the Apple device's playback engine is called the **Now Playing** list. The playlist consisting of all tracks for a given media type, such as audio or video, is called the **All Tracks** list.

4.2.3 Using Contextual Buttons

A simple remote accessory sends a `ContextButtonStatus` command to provide updated status on which buttons are held down or released. The data of the packet is a number of bytes indicating which buttons are currently held down. The bytes are constructed by ORing the masks of the buttons together. To indicate all buttons are released, the accessory must send a full data payload consisting of 0x00. While any buttons are held down, the accessory must periodically send an updated `ContextButtonStatus` packet on a 30 ms to 100 ms interval.

It is not necessary to transmit any trailing bytes in which no bits are set. If this option is exercised, the length of the packet in the header must be adjusted accordingly; that is, the packet payload length must be decreased to exclude the trailing zero byte(s) that will not be transmitted.

When the user presses and holds down a button, a simple remote accessory must generate the button status packet immediately and repeat it every 30 to 100 ms for as long as the button is pressed. If a second button is pressed while the first button is down, the button status packet sent by the accessory must include status for both buttons, and this packet must be repeated every 30 to 100 ms for as long as both buttons are held down. Table 4-14 (page 226) lists the possible Apple device button states.

The Next Track and Previous Track commands skip to the next or previous track. If the current track has played less than two seconds, Previous Track backs up to the beginning of the previous track. If the current track has played more than two seconds, it backs up to the beginning of the current track. These commands skip to the next or previous track even when the current track has chapters.

If the current track is an audiobook or a podcast with chapters, the Next Chapter and Previous Chapter commands skip to the next or previous chapter; otherwise they have no effect. If a track has chapters and has played more than two seconds of the current chapter, the Previous Chapter command backs up to the beginning of the current chapter; if the track has played less than two seconds, it backs up to the beginning of the previous chapter.

Some Apple device button states are interpreted differently by the Apple device when pressed and held down, in most cases for 2 seconds or more. These differences are the following:

- The Next Track button is treated as a Scan Forward button when pressed and held while a track is playing.
- The Previous Track button is treated as a Scan Backward button when pressed and held for 5 seconds or more while a track is playing.
- The Play/Pause button is treated as a Power Off button when pressed and held.
- In Apple devices before the 4G iPod (color display), the Menu button acted as a Display Backlight On/Off button when pressed and held. Starting with the 4G iPod (color display), pressing and holding the Menu button causes a jump to the top level menu.
- In certain older Apple devices, if the Apple device was in Browse mode the Select button was treated as an Add Track to On-The-Go Playlist button when pressed and held. This behavior is no longer supported.

Repeated Next Track and Previous Track commands (see Table 4-14 (page 226)), without an intervening button status packet indicating all buttons are up, are interpreted as Fast Forward and Rewind commands. For a locking Fast Forward or Rewind button, use the Begin Fast Forward or Begin Rewind commands to start the operation and a Play/Resume command to return to the play state.

The Next and Previous Album commands (see Table 4-14 (page 226)) have no effect if there is no next or previous album to go to in the Now Playing list. Similarly, the Next and Previous Chapter commands have no effect if the currently playing track does not have two or more chapters.

If the list of media tracks currently in the playback engine contains two or more different playlists (including the All Tracks playlist), the Next and Previous Playlist commands navigate these playlists. If the list of media tracks is from a single playlist, or none of the tracks are associated with a playlist, then the Next and Previous Playlist commands have no effect.

Use the following steps to wake an Apple device when a simple remote button is pressed (UART serial port only):

1. Send a 0xFF sync byte.
2. Wait 20 ms.
3. Send the button status packet.
4. Wait 30 to 100 ms.
5. Repeat steps 3 and 4 for as long as any button is pressed.

Multiple button status packets cannot be sent back to back; otherwise, the repeated button status packets may be misinterpreted as being part of a corrupted packet. Repeated packets must always be separated by a gap of more than 25 ms, so the Apple device knows that the new packet is not part of the previous packet (which may happen if the SYNC and SOP bytes in the first packet have been lost).

4.2.4 Using Dedicated Media Buttons

The iAP contextual button protocol sends command packets with button messages that are interpreted based on the Apple device user interface context. When an Apple device is playing media types such as images and video, however, remote controls can become overloaded and their behavior may become confusing to the Apple device user. In this case, the accessory should use dedicated media control button commands.

The dedicated media lingoes include an `iPodAck` command, so accessories know that a packet has been received, and dedicated button status commands for each media type currently supported by Apple devices: images, slideshows, videos, and audio. Use of the dedicated media lingoes requires accessory authentication.

Media control button status bits are organized so that the most frequently used buttons are assigned low bit positions. This reduces the button status packet sizes for frequently used buttons. Button status is maintained separately for all ports and all commands. This means that buttons can be in different states for different media control types.

Note: For a given port and media control type (except `ContextButtonStatus` from an accessory using the `Identify` command), if a command has not been received within approximately 200 ms after the last button status command, the button status will be reset to all buttons up.

4.2.5 Accessory Control of the iPod 5G nano Camera

This section describes how accessories can control the video camera in the iPod 5G nano, using the `CameraButtonStatus` command with General lingo notification commands.

4.2.5.1 5G nano Camera Modes

The camera in the iPod 5G nano is operated by a firmware application that can be on or off. In its Preview mode, the image that the camera captures is continuously displayed on the iPod's screen. Its Recording mode captures the video being previewed. The user can make each of three modes transition to other modes by means of user controls on either the iPod or its attached accessory, as shown in Table 4-6 (page 218).

Table 4-6 5G nano camera modes and transitions

Mode	Description	Next mode	Transition control
Camera App Off	The iPod performs only non-camera functions.	Preview	iPod only
Preview	The iPod screen displays the image that can be captured as video. In this mode and Recording mode, the iPod's music capabilities are disabled.	Recording	iPod or accessory
		Camera App Off	iPod only
Recording	The iPod is recording video	Preview	iPod or accessory

Mode	Description	Next mode	Transition control
		Camera App Off	iPod only

If a camera accessory is designed to receive feedback from the 5G nano, it must register for notifications, as specified in "Camera Accessory Identification" (page 220). After registration, the iPod notifies the accessory of its current mode and then sends notifications whenever it changes mode.

4.2.5.2 Camera Sessions

A 5G nano camera session begins when the user launches the iPod's Camera Application. The application starts in Preview mode.

When the iPod is in Preview mode the user can order it to start recording video, using either the iPod or the accessory. The iPod goes to Recording mode and records video. When the user stops recording, using either the iPod or the accessory, the iPod returns to Preview mode.

In either Preview or Recording mode the user can quit the Camera Application, using the iPod. The iPod quits the application and returns to its non-camera functionality.

The following is a typical sequence of user actions to capture a video:

1. Launch the Camera application on the iPod. This sets Preview Mode.
2. Order video recording to start, using either the iPod or the accessory.
3. Order video recording to end, using either the iPod or the accessory, or quit the Camera Application, using the iPod.

4.2.5.3 Accessory Feedback

In some situations, the user may operate the 5G nano's camera functions by means of an accessory without being able to see the iPod or otherwise get feedback from it. Thus it is desirable, but not required, for camera-control accessories to provide feedback to the user about the iPod camera's modes.

A simple feedback model might be for the accessory to provide a tristate indicator, such as a LED display, with OFF, MIDDLE, and ON states. When the accessory is connected to the iPod, and while it is going through its identification and authentication process, the indicator would be OFF. Its MIDDLE and ON states would then follow the guidelines shown in Table 4-7 (page 219).

Table 4-7 Suggested accessory feedback indications

5G nano mode	Indicator	Comments
Camera App Off	OFF	Default accessory feedback state
Preview	MIDDLE	MIDDLE state indicates Preview mode
Recording	ON	Indicator remains ON until the iPod notifies the accessory that it has gone to Preview or Camera App Off mode, or communication between the accessory and the iPod is interrupted.

4.2.5.4 Camera Accessory Identification

Every accessory that supports the 5G nano camera technology described in this document must identify and authenticate itself for at least the General and Simple Remote lingoes (Lingoes 0x00 and 0x02), as specified in "Identification" (page 93).

After it has received an `AckAccessoryAuthenticationStatus` command with a status of 0x00 from the iPod (completing the authentication process), the accessory can start sending `CameraButtonStatus` commands to it. An accessory that wants feedback from the iPod may also query the iPod's notification capabilities, using `GetSupportedEventNotification` and `RetSupportedEventNotification`, then send a `SetEventNotification` command, as described in "Camera Interface Commands" (page 220). To get camera feedback, the `SetEventNotification` command sets bit 4 (Camera Notifications) in its bitmask. The iPod replies with an `iPodNotification` command, passing the current mode of its Camera Application. Because the user can set the iPod's camera to either Preview or Recording mode, the accessory must be prepared to configure itself to be compatible with any of the modes listed in Table 4-6 (page 218).

If it has sent a `SetEventNotification` command, the accessory must be prepared to receive `iPodNotification` commands from the iPod every time the iPod's camera state changes. The accessory may use these notifications to provide feedback to the user, as suggested in "Accessory Feedback" (page 219).

4.2.5.5 Camera Interface Commands

Accessory communication with the 5G nano camera is implemented by five iAP commands, as shown in Table 4-8 (page 220).

Table 4-8 iPod camera interface commands

Lingo ID	Cmd ID	Command name	Direction	Parameters:bytes
0x00	0x4F	`GetSupportedEvent-Notification`	Acc to Dev	`{transID:2}`
0x00	0x51	`RetSupportedEvent-Notification`	Dev to Acc	`{transID:2, eventNotifyMask:8}`
0x00	0x49	`SetEventNotification`	Acc to Dev	`{transID:2, eventNotifyMask:8}`
0x00	0x4A	`iPodNotification`	Dev to Acc	`{transID:2, notificationType:1, payload:1}`
0x02	0x0E	`CameraButtonStatus`	Acc to Dev	`{transID:2, buttonStatus:1}`

4.2.5.6 Sample Command Sequences

This section describes two scenarios in which an accessory communicates with a 5G nano to control its camera. For further information about notification commands, see "Status Notifications From Apple Devices" (page 48).

4.2.5.6.1 Scenario 1: Video Control On and Off by the Accessory

A typical scenario for recording video under complete accessory control includes these actions:

1. The accessory is connected and authenticates itself. Its feedback indicator is OFF (see Table 4-7 (page 219)).
2. The accessory sends `GetSupportedEventNotification` to the iPod to determine which event notifications it supports.
3. The iPod responds with `RetSupportedEventNotification`, passing a mask that reports which notifications it supports.
4. The accessory sends `SetEventNotification` to the iPod, requesting notifications for camera and flow control events.
5. The Camera Application is not yet launched, so the iPod sends the accessory an `iPodNotification` command with a payload of Camera App Off (see Table 3-117 (page 188)).
6. Using iPod controls, the user launches the Camera application, which comes up in Preview mode.
7. The iPod sends the accessory an `iPodNotification` command with a payload of Preview.
8. The accessory changes its Indicator to the Middle state.
9. The user presses the camera button on the accessory.
10. The accessory sends a `CameraButtonStatus` command to the iPod, passing 0x01.
11. The accessory sends a second "button-up" `CameraButtonStatus` command to the iPod, passing 0x00.
12. The iPod transitions to its Recording mode and sends a corresponding notification to the accessory.
13. The accessory turns On its Indicator.
14. The iPod records video...
15. The user presses the camera button on the accessory.
16. The accessory sends a `CameraButtonStatus` command to the iPod, passing 0x01.
17. The accessory sends a second "button-up" `CameraButtonStatus` command to the iPod, passing 0x00. The iPod stops recording video.
18. The iPod transitions to its Preview mode and sends a corresponding notification to the accessory.
19. The accessory changes its Indicator to the Middle state.

As a result of this sequence, the accessory has started and ended video recording, during which time the accessory's Indicator has been On. A typical sequence of commands that implements this scenario is listed in Table 4-9 (page 222).

Table 4-9 Sample video control commands, accessory on and off

Step	Accessory command	iPod command	Comment
The accessory identifies itself as supporting the General and Simple Remote lingoes and performs the actions necessary to pass accessory authentication, as described in "Identification" (page 93).			
1		AckAccessory-AuthenticationStatus	acknowledging authentication status 'Success (OK)'
2	GetSupportedEvent-Notification		requesting supported notifications
3		RetSupportedEvent-Notification	reporting that notifications for events 'flow control' and 'camera' are supported
4	SetEventNotification		setting notifications for events 'flow control' and 'camera'
5		iPodNotification	sending notification 'Camera Not Ready'
6		iPodAck	acknowledging 'Success (OK)' to command 'General Lingo::SetEventNotification'
The user launches the Camera application, using controls on the iPod.			
7		iPodNotification	sending notification 'Preview'
The user presses the Camera Action button on the accessory.			
8	CameraButtonStatus		Camera Action
9	CameraButtonStatus		button up
10		iPodAck	acknowledging 'Success (OK)' to command 'Simple Remote Lingo::CameraButtonStatus'
11		iPodAck	acknowledging 'Success (OK)' to command 'Simple Remote Lingo::CameraButtonStatus'
12		iPodNotification	sending notification 'Recording'
The user presses the Camera Action button on the accessory.			
13	CameraButtonStatus		Camera Action
14	CameraButtonStatus		button up

Step	Accessory command	iPod command	Comment
15		iPodAck	acknowledging 'Success (OK)' to command 'Simple Remote Lingo::CameraButtonStatus'
16		iPodAck	acknowledging 'Success (OK)' to command 'Simple Remote Lingo::CameraButtonStatus'
17		iPodNotification	sending notification 'Preview'

4.2.5.6.2 Scenario 2: Video Control On by the iPod and Off by the Accessory

This scenario differs from Scenario 1 in that the iPod starts video recording and the accessory ends it. The following actions take place:

1. The accessory is connected and authenticates itself. Its feedback indicator is OFF (see Table 4-7 (page 219)).
2. The accessory sends GetSupportedEventNotification to the iPod to determine which event notifications it supports.
3. The iPod responds with RetSupportedEventNotification, passing a mask that reports which notifications it supports.
4. The accessory sends SetEventNotification to the iPod, requesting notifications for camera and flow control events.
5. The Camera Application is not yet launched, so the iPod sends the accessory an iPodNotification command with a payload of Camera App Off (see Table 3-117 (page 188)).
6. Using iPod controls, the user launches the Camera application, which starts in Preview mode.
7. The iPod sends the accessory an iPodNotification command with a payload of Preview.
8. The accessory changes its Indicator to the Middle state.
9. The user starts Recording using controls on the iPod.
10. The iPod sends the accessory an iPodNotification command with a payload of Recording.
11. The accessory turns On its Indicator.
12. The iPod records video...
13. The user presses the camera button on the accessory.
14. The accessory sends a CameraButtonStatus command to the iPod, passing 0x01.
15. The accessory sends a second “button-up” CameraButtonStatus command to the iPod, passing 0x00. The iPod stops recording video.
16. The iPod transitions to its Preview mode and sends a corresponding notification to the accessory.

17. The accessory changes its Indicator to the Middle state.

As a result of this sequence, the iPod has started and the accessory has ended a video recording, during which time the accessory's Indicator has been On. A typical sequence of commands that implements this scenario is listed in Table 4-10 (page 224).

Table 4-10 Sample video control commands, iPod on and accessory off

Step	Accessory command	iPod command	Comment
The accessory identifies itself as supporting the General and Simple Remote lingoes and performs the actions necessary to pass accessory authentication, as described in "Identification" (page 93).			
1		`AckAccessory-AuthenticationStatus`	acknowledging authentication status 'Success (OK)'
2	`GetSupportedEvent-Notification`		requesting supported notifications
3		`RetSupportedEvent-Notification`	reporting that notifications for events 'flow control' and 'camera' are supported
4	`SetEventNotification`		setting notifications for events 'flow control' and 'camera'
5		`iPodNotification`	sending notification 'Camera Not Ready'
6		`iPodAck`	acknowledging 'Success (OK)' to command 'General Lingo::SetEventNotification'
The user launches the Camera application, using controls on the iPod.			
7		`iPodNotification`	sending notification 'Preview'
The user starts recording video, using controls on the iPod.			
8		`iPodNotification`	sending notification 'Recording'
The user presses the Camera Action button on the accessory.			
9	`CameraButtonStatus`		Camera Action
10	`CameraButtonStatus`		button up
11		`iPodAck`	acknowledging 'Success (OK)' to command 'Simple Remote Lingo::CameraButtonStatus'
12		`iPodAck`	acknowledging 'Success (OK)' to command 'Simple Remote Lingo::CameraButtonStatus'

Step	Accessory command	iPod command	Comment
13		iPodNotification	sending notification 'Preview'

4.2.6 USB Human Interface Device Reports

Four commands in the Simple Remote lingo support the registration and exchange of USB Human Interface Device (HID) reports. These commands are listed in Table 4-11 (page 225).

Table 4-11 USB HID commands

Cmd ID	Command name	Direction	Parameters:bytes
0x0F	RegisterDescriptor	Acc to Dev	{index:1, VID:2, PID:2, country_code:1, descriptor:<var>}
0x10	iPodHIDReport	Acc to Dev	{index:1, report_type:1, report:<var>}
0x11	AccessoryHIDReport	Dev to Acc	{index:1, report_type:1, report:<var>}
0x12	UnregisterDescriptor	Acc to Dev	{index:1}

The contents of the HID reports that an accessory may exchange with an Apple device are determined by the USB *Device Class Definition for HID* specification. The HID reports are sent as opaque byte streams. Certain HID Consumer Page (0x0C) usages are also supported by iOS in addition to the standard HID alphanumeric keycodes. These controls are listed in Table 4-12 (page 225) and are described in detail in the USB *HID Usage Tables* specification, Version 1.12.

Note: To control display brightness, accessories must use the iAP Display Remote lingo commands GetiPodStateInfo and SetiPodStateInfo.

Table 4-12 USB HID Consumer Page controls supported by iOS

HID Usage ID	HID Usage Name	iOS Function
0x0030	Power	Lock
0x0040	Menu	Home
0x00B5	Scan Next Track	Transport Right
0x00B6	Scan Previous Track	Transport Left
0x00CD	Play/Pause	Play/Pause
0x00E2	Mute	Mute
0x00E9	Volume Increment	Louder
0x00EA	Volume Decrement	Softer

HID Usage ID	HID Usage Name	iOS Function
0x01AE	AL Keyboard Layout	Toggle Onscreen Keyboard
0x01B1	AL Screen Saver	Picture Frame
0x0221	AC Search	Spotlight

Note: The glyphs that the accessory displays on its physical keycaps must be approved by Apple MFi Licensing.

4.2.7 Command 0x00: ContextButtonStatus

Lingo: 0x02 — Origin: Accessory

The accessory sends this command to the Apple device when a button event occurs. The button status is a bitmask representing each button that is currently pressed. The accessory must send the button status packet repeatedly at intervals between 30 and 100 ms, while one or more buttons are pressed. When all buttons are released, the accessory must send a button status packet with a 0x00 payload to indicate that no buttons are pressed. The Apple device does not return a packet to the accessory in response to this command.

Table 4-13 `ContextButtonStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Byte index 0, button states 7:0; see Table 4-14 (page 226) for a list of button states recognized by Apple devices.
0x*NN*	1	Byte index 1, button states 15:8 (optional)
0x*NN*	1	Byte index 2, button states 23:16 (optional)
0x*NN*	1	Byte index 3, button states 31:24 (optional)
Packet footer specified in "Command Packets" (page 109).		

Table 4-14 (page 226) lists the available buttons and their bitmasks.

Note: Touchscreen-equipped devices support only a subset of this list, as specified in "Interaction with iOS Media Applications" (page 54).

Table 4-14 Button states

Button name	Number	Byte index	Button bitmask
Play/Pause	0	0x00	0x01

Button name	Number	Byte index	Button bitmask
Volume Up	1	0x00	0x02
Volume Down	2	0x00	0x04
Next Track	3	0x00	0x08
Previous Track	4	0x00	0x10
Next Album	5	0x00	0x20
Previous Album	6	0x00	0x40
Stop	7	0x00	0x80
Play/Resume	8	0x01	0x01
Pause	9	0x01	0x02
Mute toggle	10	0x01	0x04
Next Chapter	11	0x01	0x08
Previous Chapter	12	0x01	0x10
Next Playlist	13	0x01	0x20
Previous Playlist	14	0x01	0x40
Shuffle Setting Advance	15	0x01	0x80
Repeat Setting Advance	16	0x02	0x01
Power On	17	0x02	0x02
Power Off	18	0x02	0x04
Backlight for 30 Seconds	19	0x02	0x08
Begin Fast Forward	20	0x02	0x10
Begin Rewind	21	0x02	0x20
Menu	22	0x02	0x40
Select	23	0x02	0x80
Up Arrow	24	0x03	0x01
Down Arrow	25	0x03	0x02
Backlight Off	26	0x03	0x04
Reserved	31:27	0x03	0xF8

Notes

- The Begin Fast Forward and Begin Rewind states must be sent every 30 to 100 ms to assure continuous fast forward or rewind actions.
- If the user holds down the Fast Forward button (sending repeated Fast Forward states) through a track change, the new track begins playing at normal speed.
- The Menu button navigates only within iPod/Music/Video applications and does not return the user to the Home screen.

4.2.8 Command 0x01: iPodAck

Lingo: 0x02 — Origin: Apple device

The Apple device sends this command in response to any command sent from the accessory, except command 0x00. an `iPodAck` response is sent when a command that does not return any data has completed, when a bad parameter is received, or when an unsupported or invalid command is received.

Table 4-15 `iPodAck` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command status; see Table 3-6 (page 125).
0x*NN*	1	ID of the command being acknowledged
Packet footer specified in "Command Packets" (page 109).		

4.2.9 Command 0x03: VideoButtonStatus

Lingo: 0x02 — Origin: Accessory

The accessory sends this command to the Apple device when a video-specific button event occurs. The button status is a bitmask representing each button that is currently pressed. The button status packet must be repeatedly sent by the accessory at intervals between 30 ms and 100 ms while one or more buttons are pressed. When all buttons are released, the accessory must send a button status packet with a 0x00 payload to indicate that no buttons are pressed. In response, the Apple device will return an `iPodAck` packet containing the command status to the accessory.

Note: Apple products running iOS 3.2 support only the Stop, Play/Resume, and Pause button values.

Table 4-16 `VideoButtonStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		

Value	Bytes	Parameter
0x*NN*	1	Byte index 0, video-specific button states 7:0; see Table 4-17 (page 229) for a list of video-specific button states recognized by Apple devices.
0x*NN*	1	Byte index 1, button states 15:8 (optional)
0x*NN*	1	Byte index 2, button states 23:16 (optional)
0x*NN*	1	Byte index 3, button states 31:24 (optional)
Packet footer specified in "Command Packets" (page 109).		

Table 4-17 Video-specific button values

Button name	Number	Byte index	Button bitmask
Play/Pause	0	0x00	0x01
Next video	1	0x00	0x02
Previous video	2	0x00	0x04
Stop	3	0x00	0x08
Play/Resume	4	0x00	0x10
Pause	5	0x00	0x20
Begin FF	6	0x00	0x40
Begin REW	7	0x00	0x80
Next chapter	8	0x01	0x01
Previous chapter	9	0x01	0x02
Next frame	10	0x01	0x04
Previous frame	11	0x01	0x08
Reserved	12:15	0x01	0xF0
Reserved	16:23	0x02	0xFF
Reserved	24:31	0x03	0xFF

The Next Video and Previous Video button commands skip to the next or previous video. If the current video has played less than two seconds, Previous Video backs up to the beginning of the previous video; if it has played more than two seconds, it backs up to the beginning of the current video. These commands skip to the next or previous video even when the current video has chapters. If the current video has chapters, the Next Chapter and Previous Chapter commands skip to the next or previous chapter; otherwise they have no

effect. If the video has chapters and has played more than two seconds of the current chapter, the Previous Chapter command backs up to the beginning of the current chapter; if the track has played less than two seconds, it backs up to the beginning of the previous chapter.

4.2.10 Command 0x04: AudioButtonStatus

Lingo: 0x02 — Origin: Accessory

The accessory sends this command to the Apple device when an audio-specific button event occurs. The button status is a bitmask representing each button that is currently pressed. The button status packet must be repeatedly sent by the accessory at intervals between 30 ms and 100 ms while one or more buttons are pressed. When all buttons are released, the accessory must send a button status packet with a 0x00 payload to indicate that no buttons are pressed. In response, the Apple device will return to the accessory an `iPodAck` message containing the command status.

Note: Apple products running iOS 3.2 support only the Stop, Play/Resume, and Pause button values.

Table 4-18 `AudioButtonStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Byte index 0, audio-specific button states 7:0; see Table 4-19 (page 230) for a list of audio-specific button states recognized by Apple devices.
0x*NN*	1	Byte index 1, button states 15:8 (optional)
0x*NN*	1	Byte index 2, button states 23:16 (optional)
0x*NN*	1	Byte index 3, button states 31:24 (optional)
Packet footer specified in "Command Packets" (page 109).		

Table 4-19 Audio-specific button values

Button name	Number	Byte index	Button bitmask
Play/Pause	0	0x00	0x01
Volume Up	1	0x00	0x02
Volume Down	2	0x00	0x04
Next Track	3	0x00	0x08
Previous Track	4	0x00	0x10
Next Album	5	0x00	0x20
Previous Album	6	0x00	0x40

Button name	Number	Byte index	Button bitmask
Stop	7	0x00	0x80
Play/Resume	8	0x01	0x01
Pause	9	0x01	0x02
Mute toggle	10	0x01	0x04
Next chapter	11	0x01	0x08
Previous chapter	12	0x01	0x10
Next playlist	13	0x01	0x20
Previous playlist	14	0x01	0x40
Shuffle setting advance	15	0x01	0x80
Repeat setting advance	16	0x02	0x01
Begin FF	17	0x02	0x02
Begin REW	18	0x02	0x04
Record	19	0x02	0x08
Reserved	20:23	0x02	0xF0
Reserved	24:31	0x03	0xFF

The Next and Previous Track, Album, and Playlist button commands skip to the next or previous track, album, or playlist. If the current track, album, or playlist has played less than two seconds, the Previous command backs up to the beginning of the previous one; if it has played more than two seconds, it backs up to the beginning of the current one. These commands skip to the next or previous track, album, or playlist even when the current one has chapters. If the current track has chapters, the Next Chapter and Previous Chapter commands skip to the next or previous chapter; otherwise they have no effect. If the track has chapters and has played more than two seconds of the current chapter, the Previous Chapter command backs up to the beginning of the current chapter; if the track has played less than two seconds, it backs up to the beginning of the previous chapter.

4.2.11 Command 0x0B: iPodOutButtonStatus

Lingo: 0x02 — Origin: Accessory

WARNING: The Apple device must be in iPod Out mode when the accessory sends this command.

The accessory sends this command to inform the Apple device that the user has pushed one or more accessory buttons intended to control the Apple device. The `iPodOutButtonMask` field is a bitmask representing each button that is currently pressed. The accessory should repeatedly send the packet shown in Table 4-20 (page 232) at an interval of 30 to 100 ms while one or more buttons are pressed (note that the bytes containing bits

8 to 31 of the `iPodOutButtonMask` parameter are optional). When all buttons are released, the accessory must send a `iPodOutButtonStatus` command with a packet length of 0x04 and a `iPodOutButtonMask` value of 0x0.

In response to each packet, the Apple device returns a Simple Remote `iPodAck` command with the command status. The button events reported by this command are handled by the Apple device in a context-specific manner.

Table 4-20 `iPodOutButtonStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`buttonSource`: Location of button; see Table 4-21 (page 232).
0x*NN*	1	`iPodOutButtonMask`: Button status bitmask, bits 7:0; see Table 4-22 (page 232).
0x*NN*	1	`iPodOutButtonMask`: Button status bitmask; optional bits 15:8
0x*NN*	1	`iPodOutButtonMask`: Button status bitmask; optional bits 23:16
0x*NN*	1	`iPodOutButtonMask`: Button status bitmask; optional bits 31:24
Packet footer specified in "Command Packets" (page 109).		

Table 4-21 iPod Out control locations

Value	Meaning
0x00	Car center console
0x01	Steering wheel
0x02	Car dashboard
0x03-0xFF	Reserved

Table 4-22 iPod Out button status values

Bit	Meaning
00	Select event
01	Left event
02	Right event
03	Up event
04	Down event
05	Menu button event

Bit	Meaning
06-07	Reserved

4.2.12 Command 0x0C: RotationInputStatus

Lingo: 0x02 — Origin: Accessory

WARNING: The Apple device must be in iPod Out mode when the accessory sends this command.

The accessory sends this command to inform the Apple device that the user is acting on a rotating device (such as a control knob with detents or a free-turning wheel) intended to control the Apple device. The accessory should send command packets at intervals of no more than 100 ms while the user action is in progress, and then it must send a final packet when the user action has finished.

Note: If the accessory does not send packets at least every 100 ms, the Apple device may not respond as expected. However, the Apple device should respond properly to a late packet if it contains a correct `userActionDurationMs` value.

If the user has turned the rotating device to a stationary position and is holding it in place (assuming the accessory can detect such an action), this command must be sent with a `rotationAction` value of 0x02 at intervals of no more than 100 ms while the user is holding the input device in place. Once the user stops acting on the rotating device, the accessory must send this command with a `rotationAction` value of 0x00.

Table 4-23 `RotationInputStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	`userActionDurationMs`: Number of milliseconds since the start of the current user action.
0x*NN*	1	`rotationSource`: Location of wheel; see Table 4-21 (page 232).
0x*NN*	1	`controllerType`: Type of wheel; see Table 4-24 (page 234).
0x*NN*	1	`rotationDirection`: Direction of rotation; see Table 4-25 (page 234).
0x*NN*	1	`rotationAction`: Rotation action; see Table 4-26 (page 234).
0x*NN*	1	`rotationType`: Type of rotation; see Table 4-27 (page 234).
0x*NN*	2	`detentsOrDegreesMoved`: Number of detents or degrees the user has moved the wheel since the last `RotationInputStatus` packet
0x*NN*	2	`detentsOrDegreesTotal`: Constant number of detents or degrees in a full turn of the wheel (maximum = 360)

Value	Bytes	Parameter
Packet footer specified in "Command Packets" (page 109).		

Table 4-24 Wheel types

Value	Meaning
0x00	Free wheel
0x01-0xFF	Reserved

Table 4-25 Rotation directions

Value	Meaning
0x00	Counterclockwise (left)
0x01	Clockwise (right)
0x02-0xFF	Reserved

Table 4-26 Rotation actions

Value	Meaning
0x00	Rotation action completed; user has released control
0x01	Rotation in progress; wheel turning
0x02	Rotation repeat; the user has not advanced the wheel from the position reported in the last `RotationInputStatus` packet
0x03-0xFF	Reserved

Table 4-27 Rotation types

Value	Meaning
0x00	Wheel has detents
0x01	Wheel reports angular degrees
0x02-0xFF	Reserved

4.2.12.1 Examples of Using RotationInputStatus

Following are two examples of `RotationInputStatus` packets sent by an accessory.

4.2.12.1.1 Example with a Detent Wheel

- The user begins to turn wheel; the accessory sends a `RotationInputStatus` command to the Apple device after 100 ms
  - userActionDurationMs = 0
  - buttonSource = car center console (0x00)
  - controllerType = free wheel (0x00)
  - rotationDirection = right (0x01)
  - rotationAction = rotation started (0x01)
  - rotationType = detents (0x00)
  - detentsMoved = 2 (0x02)
  - detentsTotal = 16 (0x10)

- The user continues to turn wheel until 100 ms after starting rotation. During that time, the accessory sends a second `RotationInputStatus` command to the Apple device
  - userActionDurationMs = 100
  - buttonSource = car center console (0x00)
  - controllerType = free wheel (0x00)
  - rotationDirection = right (0x01)
  - rotationAction = rotation in progress (0x01)
  - rotationType = detents (0x00)
  - detentsMoved = 3 detents since last packet (0x03)
  - detentsTotal = 16 (0x10)

- The user stops after 200 ms; the accessory sends a `RotationInputStatus` command to the Apple device indicating completion
  - userActionDurationMs = 100
  - buttonSource = car center console (0x00)
  - controllerType = free wheel (0x00)
  - rotationDirection = right (0x01)
  - rotationAction = rotation action completed (0x00)
  - rotationType = detents (0x00)
  - detentsMoved = 0 detents since last packet (0x00)
  - detentsTotal = 16 (0x10)

4.2.13 Command 0x0D: RadioButtonStatus

Lingo: 0x02 — Origin: Accessory

The accessory sends this command to a 5G nano iPod to initiate a tagging action by the iPod's Radio Application. Once the accessory has authenticated itself for the Simple Remote lingo, it may send this command at any time.

Currently the only payloads for this command are the values 0x02 and 0x00 in `buttonStatus`. Any other values will result in the iPod returning an error value of 0x04 (Bad Parameter). The accessory should repeatedly send this command with a payload of 0x02 at intervals of 30-100 ms while its Radio Tag button is held down. When the button is released it must send a payload of 0x00 to indicate that no buttons are down. The iPod responds with an `iPodAck` command passing success (0x00).

This command is intended to be used with the `GetSupportedEventNotification` and `SetEventNotification` commands; see "Status Notifications From Apple Devices" (page 48).

Table 4-28 `RadioButtonStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`buttonStatus`: 0x02 = Radio button pushed to tag current song; 0x00 = button released.
Packet footer specified in "Command Packets" (page 109).		

4.2.14 Command 0x0E: CameraButtonStatus

Lingo: 0x02 — Origin: Accessory

The accessory sends this command to the Apple device to initiate changes in the Apple device's Camera mode. Once the accessory has authenticated itself for the Simple Remote lingo, it may send this command at any time.

The accessory must send `CameraButtonStatus` commands in pairs: the first command must pass 0x01 in `buttonStatus` and the second command must pass 0x00. If the accessory doesn't send a second command, the Apple device assumes such a return after a 200 ms timeout. Each `CameraButtonStatus` command may pass only the value 0x00 or 0x01; otherwise, the Apple device will acknowledge it with a Bad Parameter error status. The Apple device will acknowledge each valid `CameraButtonStatus` command with a General lingo `iPodAck` command reporting Status OK (0x00).

This command is intended to be used with the `GetSupportedEventNotification` and `SetEventNotification` commands; see "Status Notifications From Apple Devices" (page 48).

Table 4-29 `CameraButtonStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`buttonStatus`: (0x01 = down, 0x00 = up)
Packet footer specified in "Command Packets" (page 109).		

4.2.15 Command 0x0F: RegisterDescriptor

Lingo: 0x02 — Origin: Accessory

The accessory sends this command to register a USB Human Interface Device (HID) Report Descriptor for subsequent use by the `iPodHIDReport` and `AccessoryHIDReport` commands. The Apple device creates and registers a new USB device in the iPod HID layer without parsing the descriptor. The Apple device responds to this command by sending a Simple Remote `iPodAck` command.

For a description of the process of sending HID reports via iAP, see "USB Human Interface Device Reports" (page 225).

Table 4-30 `RegisterDescriptor` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`index`: USB descriptor index
0x*NN*	2	`VID`: Vendor ID of the accessory; see **Note** below.
0x*NN*	2	`PID`: Product ID of the accessory; see **Note** below.
0x*NN*	1	`country_code`: Country code of the accessory; see Table 4-31 (page 237). Localized devices, such as keyboards, should pass a country code. Nonlocalized devices must pass 0x00.
0x*NN*	1	USB HID descriptor
Packet footer specified in "Command Packets" (page 109).		

Note: The Vendor ID and Product ID parameters of the `RegisterDescriptor` command must comply with the VID/PID requirements specified in "Using an Apple Device as a USB Host" in *MFi Accessory Hardware Specification*.

Table 4-31 Country codes

Decimal	Hexadecimal	Country or localization
0	00	not supported / undefined
1	01	Arabic
2	02	Belgian
3	03	Canadian-Bilingual
4	04	Canadian-French
5	05	Czech Republic

Decimal	Hexadecimal	Country or localization
6	06	Danish
7	07	Finnish
8	08	French
9	09	German
10	0A	Greek
11	0B	Hebrew
12	0C	Hungarian
13	0D	International (ISO)
14	0E	Italian
15	0F	Japan (Katakana)
16	10	Korean
17	11	Latin American
18	12	Netherlands/Dutch
19	13	Norwegian
20	14	Persian (Farsi)
21	15	Poland
22	16	Portuguese
23	17	Russian
24	18	Slovakia
25	19	Spanish
26	1A	Swedish
27	1B	Swiss/French
28	1C	Swiss/German
29	1D	Switzerland
30	1E	Taiwan
31	1F	Turkish-Q
32	20	UK

Decimal	Hexadecimal	Country or localization
33	21	US
34	22	Croatian
35	23	Turkish-F
36–249	24–F9	reserved
250	FA	Thai
251	FB	Flemish
252	FC	Romanian
253	FD	Bulgarian
254	FE	Chinese (Simplified)

4.2.16 Command 0x10: iPodHIDReport

Lingo: 0x02 — Origin: Accessory

The accessory sends this command to report USB Human Interface Device (HID) events to the Apple device, such as changes in its user controls. The Apple device routes the report to the iPod HID layer without parsing it. The Apple device responds to this command by sending a Simple Remote `iPodAck` command.

Table 4-32 `iPodHIDReport` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`index`: USB descriptor index
0x00	1	`report_type`: Report type = input.
0x*NN*	1	USB HID report
Packet footer specified in "Command Packets" (page 109).		

4.2.17 Command 0x11: AccessoryHIDReport

Lingo: 0x02 — Origin: Apple device

The Apple device sends this command to report USB Human Interface Device (HID) events to accessory, such as requests to change its LED display. The Apple device forwards outbound reports from its HID layer to the accessory without parsing them. The accessory must respond to this command by sending the Apple device a Simple Remote `AccessoryAck` command.

Table 4-33 `AccessoryHIDReport` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`index`: USB descriptor index
0x01	1	`report_type`: Report type = output.
0x*NN*	1	USB HID report
Packet footer specified in "Command Packets" (page 109).		

4.2.18 Command 0x12: UnregisterDescriptor

Lingo: 0x02 — Origin: Accessory

The accessory sends this command to inform the Apple device that the USB HID report descriptor created by an earlier `RegisterDescriptor` command is no longer needed and may be discarded. The Apple device responds to this command by sending a Simple Remote `iPodAck` command.

Table 4-34 `UnregisterDescriptor` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`index`: USB descriptor index
Packet footer specified in "Command Packets" (page 109).		

4.2.19 Command 0x13: VoiceOverEvent

Lingo: 0x02 — Origin: Accessory

The accessory sends this command to notify the Apple device that the user has generated an interface event. The `eventType` parameter identifies the type of the event; based on this type, there may be data in the `eventData` parameter.

Note: The "Scroll Left Page" and "Scroll Right Page" events (`eventType` 0x05 and 0x06) are used by the iBook app.

Table 4-35 `VoiceOverEvent` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		

Value	Bytes	Parameter
0x*NN*	1	`eventType`: see Table 4-36 (page 241).
0x*NN*	*NN*	`eventData`: see Table 4-36 (page 241).
Packet footer specified in "Command Packets" (page 109).		

Table 4-36 `VoiceOverEvent` events and data

eventType	Event	eventData
0x00	Move to a point on the screen. This event assumes a screen size of 65536 by 65536, with both x and y values between 0 and 65535. Point (0,0) is the top left corner of the view, regardless of the orientation of the Apple device.	Byte 1: X coordinate (bits 15:8) Byte 2: X coordinate (bits 7:0) Byte 3: Y coordinate (bits 15:8) Byte 4: Y coordinate (bits 7:0)
0x01	Move To First	No event data
0x02	Move To Last	No event data
0x03	Move To Next	No event data
0x04	Move To Previous	No event data
0x05	Scroll Left Page	No event data
0x06	Scroll Right Page	No event data
0x07	Scroll Up Page	No event data
0x08	Scroll Down Page	No event data
0x09	Scroll to a point on the screen. This event assumes a screen size of 65536 by 65536, with both x and y values between 0 and 65535. Point (0,0) is the top left corner of the view, regardless of the orientation of the Apple device.	Byte 1: X coordinate (bits 15:8) Byte 2: X coordinate (bits 7:0) Byte 3: Y coordinate (bits 15:8) Byte 4: Y coordinate (bits 7:0)
0x0A	Send text from the accessory to the input field of the Apple device. Text may be sent in multiple sections, each preceded by its section index and the index of the last section of text. If the text fits into one command, both of these numbers must be 0x00. Multiple text sections must be sent as consecutive `VoiceOverEvent` commands with the same transaction ID. The whole text, with all its sections concatenated, must be a single null-terminated UTF-8 string.	Byte 1: current section index (bits 15:8) Byte 2: current section index (bits 7:0) Byte 3: last section index (bits 15:8) Byte 4: last section index (bits 7:0) Bytes 5-*NN*: UTF-8 characters
0x0B	Cut	No event data
0x0C	Copy	No event data

eventType	Event	eventData
0x0D	Paste	No event data
0x0E	Home	No event data
0x0F	Send a touch event to the currently selected VoiceOver item.	Bytes 1-4: Reserved; set to 0x00000000 Byte 5: Touch event type: 0x00: Began 0x01: Moved 0x02: Stationary 0x03: Ended 0x04: Canceled 0x05-0xFF: Reserved
0x10	Set scale display factor (magnification)	A 16-bit unsigned integer: Byte 1: Scale factor (bits 15:8) Byte 2: Scale factor (bits 7:0)
0x11	Center display around a point on the screen. This event assumes a screen size of 65536 by 65536, with both x and y values between 0 and 65535. Point (0,0) is the top left corner of the view, regardless of the orientation of the Apple device.	Byte 1: X coordinate (bits 15:8) Byte 2: X coordinate (bits 7:0) Byte 3: Y coordinate (bits 15:8) Byte 4: Y coordinate (bits 7:0)
0x12	Pause speaking	No event data
0x13	Resume speaking	No event data
0x14	Read all text from current point	No event data
0x15	Read all text from top	No event data
0x16	Send text from the accessory to an iPhone, to be spoken to the user by the iPhone's text-to-speech engine. Text may be sent in multiple sections, each preceded by its section index and the index of the last section of text. If the text fits into one command, both of these numbers must be 0x00. Multiple text sections must be sent as consecutive `VoiceOverEvent` commands with the same transaction ID. The whole text to be spoken, with all its sections concatenated, must be a single null-terminated UTF-8 string.	Byte 1: current section index (bits 15:8) Byte 2: current section index (bits 7:0) Byte 3: last section index (bits 15:8) Byte 4: last section index (bits 7:0) Bytes 5-*NN*: UTF-8 characters
0x17	Escape; lets VoiceOver exit from a modal view, such as an alert or popover.	No event data
0x18-0xFF	Reserved	

4.2.20 Command 0x14: GetVoiceOverParameter

Lingo: 0x02 — Origin: Accessory

The accessory sends this command to the Apple device to get the value of one of the Apple device's VoiceOver parameters.

Table 4-37 `GetVoiceOverParameter` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`paramType`: The type of parameter to be returned; see Table 4-39 (page 243).
Packet footer specified in "Command Packets" (page 109).		

4.2.21 Command 0x15: RetVoiceOverParameter

Lingo: 0x02 — Origin: Apple device

The Apple device sends this command to the accessory, in response to a `GetVoiceOverParameter` command, to return the value of one of the Apple device's VoiceOver parameters.

Table 4-38 `RetVoiceOverParameter` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`paramType`: the parameter type requested by `GetVoiceOverParameter`; see Table 4-39 (page 243).
0x*NN*	1	`paramValue`: the value of the requested parameter; see Table 4-39 (page 243).
Packet footer specified in "Command Packets" (page 109).		

Table 4-39 VoiceOver parameter types and values

Type ID	Parameter	Value	Notes
0x00	VoiceOver volume	An 8-bit unsigned integer between 0 (mute) and 255 (full volume).	
0x01	Speaking rate	An 8-bit unsigned integer between 0 (slowest) and 255 (fastest). 127 is normal.	

Type ID	Parameter	Value	Notes
0x02	VoiceOver enabled	An 8 bit unsigned integer that encodes a Boolean. Zero = false, nonzero = true.	Cannot be set by `SetVoiceOverParameter`; use `SetiPodPreferences`. See Table 3-57 (page 152), Class 0x14.
0x03-0xFF	Reserved		

4.2.22 Command 0x16: SetVoiceOverParameter

Lingo: 0x02 — Origin: Accessory

The accessory sends this command to set the value of one of the Apple device's VoiceOver parameters.

Table 4-40 `SetVoiceOverParameter` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`paramType`: the VoiceOver parameter type to be set; see Table 4-39 (page 243). Type 0x02, VoiceOver enabled, cannot be set.
0x*NN*	1	`paramValue`: the value of the parameter to be set; see Table 4-39 (page 243).
Packet footer specified in "Command Packets" (page 109).		

4.2.23 Command 0x17: GetCurrentVoiceOverItemProperty

Lingo: 0x02 — Origin: Accessory

The accessory sends this command to the Apple device to get a specific property of the item currently selected in the Apple device's user interface.

Note: This command may cause `RetCurrentVoiceOverItemProperty` to return large amounts of data. If necessary, the accessory may cancel the whole transaction by passing the ID of `GetCurrentVoiceOverItemProperty` to the General lingo `Cancel` command (0x50).

Table 4-41 `GetCurrentVoiceOverItemProperty` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`propertyType`: The type of property to be returned; see Table 4-43 (page 245).
Packet footer specified in "Command Packets" (page 109).		

4.2.24 Command 0x18: RetCurrentVoiceOverItemProperty

Lingo: 0x02 — Origin: Apple device

The Apple device sends this command to the accessory, in response to a `GetCurrentVoiceOverItemProperty` command or asynchronously, to return the value of one of the properties of the Apple device's currently selected user interface item.

Table 4-42 `RetCurrentVoiceOverItemProperty` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`propertyType`: the property type requested by `GetCurrentVoiceOverItemProperty`; see Table 4-43 (page 245).
0x*NN*	1	`propertyValue`: the value of the requested property; see Table 4-43 (page 245).
Packet footer specified in "Command Packets" (page 109).		

Table 4-43 Currently selected item property types and values

ID	Property	Value
0x00	Label	A null-terminated UTF-8 string. The string may be sent in multiple sections, each preceded by its section index and the index of the last section of text. If the whole string fits into one command, both of these numbers must be 0x00. Multiple sections of the string must be sent as consecutive `RetCurrentVoiceOverItemProperty` commands with the same transaction ID. Byte 8: current section index (bits 15:8) Byte 9: current section index (bits 7:0) Byte 10: last section index (bits 15:8) Byte 11: last section index (bits 7:0) Bytes 12-*NN*: UTF-8 characters
0x01	Value	
0x02	Hint	
0x03	Frame	The enclosing rectangle of the currently selected item. The graphics environment assumes a screen size of 65536 by 65536, with both x and y values between 0 and 65535. Point (0,0) is the top left corner of the view, regardless of the orientation of the Apple device. Byte 8: top left X coordinate (bits 15:8) Byte 9: top left X coordinate (bits 7:0) Byte 10: top left Y coordinate (bits 15:8) Byte 11: top left Y coordinate (bits 7:0) Byte 12: bottom right X coordinate (bits 15:8) Byte 13: bottom right X coordinate (bits 7:0) Byte 14: bottom right Y coordinate (bits 15:8) Byte 15: bottom right Y coordinate (bits 7:0)

ID	Property	Value
0x04	Traits	Traits are attributes of the selected item. Zero or more traits may be returned as a list of 16-bit big endian values. The number of traits returned equals the length of the packet minus 3 and divided by 2. Currently defined traits and their values are the following: Button: 0x0000 Link: 0x0001 Search Field: 0x0002 Image: 0x0003 Selected: 0x0004 Sound: 0x0005 Keyboard Key: 0x0006 Static Text: 0x0007 Summary Element: 0x0008 Not Enabled: 0x0009 Updates Frequently: 0x000A Starts Media Session: 0x000B Adjustable: 0x000C Back Button: 0x000D Map: 0x000E Delete Key: 0x000F

4.2.25 Command 0x19: SetVoiceOverContext

Lingo: 0x02 — Origin: Accessory

The accessory sends this command to the Apple device to set the Apple device's current user interface context.

Table 4-44 `SetVoiceOverContext` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`paramType`: The user interface context to be set; see Table 4-45 (page 246).
Packet footer specified in "Command Packets" (page 109).		

Table 4-45 User interface contexts

ID	Context	Description
0x00	None	There is no current context.
0x01	Header	The user is working in a header.

ID	Context	Description
0x02	Link	The user is executing a link.
0x03	Form	The user is filling out a form.
0x04	Cursor	The user is manipulating the cursor.
0x05	Vertical navigation	Sets MoveNext/MovePrevious to move down/up to the next item vertically.
0x06	Value adjustment	Sets MoveNext/MovePrevious to increment/decrement the value of the current item by 5%.
0x07	Zoom adjustment	Sets MoveNext/MovePrevious to zoom the current item in/out by 1 increment.
0x08-0xFF	Reserved	

4.2.26 Command 0x1A: VoiceOverParameterChanged

Lingo: 0x02 — Origin: Apple device

The Apple device sends this command to the accessory whenever its VoiceOver feature is enabled or disabled, either because the accessory changed the VoiceOver Preference or because the user changed this setting on the Apple device. Note that user changes may also be done through another VoiceOver-capable accessory.

This notification is sent to any connected accessory that has set its VoiceOver capability bit (0x11) when authenticating via IDPS. Commands sent to the accessory after it has received a notification that VoiceOver is no longer enabled, but before it has received a notification that VoiceOver is enabled, may be ignored.

Table 4-46 `VoiceOverParameterChanged` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`paramType`: the parameter type that has changed; see Table 4-39 (page 243).
0x*NN*	1	`paramValue`: the new value of the changed parameter; see Table 4-39 (page 243).
Packet footer specified in "Command Packets" (page 109).		

4.2.27 Command 0x81: AccessoryAck

Lingo: 0x02 — Origin: Accessory

The accessory sends this command in response to a command received from the Apple device.

Table 4-47 `AccessoryAck` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command status: 0x00 if the accessory handled the Apple device command successfully.
0x*NN*	1	ID of the command being acknowledged
Packet footer specified in "Command Packets" (page 109).		

4.3 Lingo 0x03: Display Remote Lingo

The Display Remote lingo is for accessory devices that need to control the state of the Apple device, display information about the state of the Apple device on a remote display, or control the state of the Apple device equalizer. The Display Remote protocol can be used by simple inline-display remotes (remotes that have single-line display and play control buttons) and more complex devices that have full multiline graphical displays to show information about the track, artist, or album; current play or pause state; track position; battery; shuffle and time.

For example, the Display Remote protocol can be used in an automotive application to show currently playing track information on the in-vehicle display while allowing the user to browse the database stored in the Apple device. Such an application could let the user choose where to browse the database (on the in-vehicle display or on the Apple device) and switch between Extended Interface and Display Remote modes, depending on the setting. For a sample command sequence that declares this lingo, see Table 4-294 (page 391).

By supporting multiple lingoes, an accessory can use the Display Remote lingo in combination with other lingoes to create a fully functional device/accessory system. Accessories can also use this lingo to control the state of the Apple device equalizer. The Display Remote lingo supports serial accessories attached to the 30-pin connector.

The Display Remote command set uses a single byte command format similar to the General and Simple Remote lingoes. Devices using the Display Remote lingo can identify using the General lingo (0x0) command `IdentifyDeviceLingoes` (0x13). See "Command 0x13: IdentifyDeviceLingoes" (page 133) for more information.

Note: The Display Remote lingo is an authenticated lingo, with some exceptions. USB accessories must authenticate before they can use the Display Remote lingo. Serial accessories have access only to the equalizer control, battery state, and sound check state commands (commands 0x01–0x07 and 0x1A–0x1E) if they do not authenticate. Please refer to Table 4-48 (page 249) to determine which commands require authentication.

The Display Remote lingo can operate in notification (interrupt) mode, where the Apple device sends event notifications to the accessory, or in polled (non-interrupt) mode. In polled mode, the accessory should send requests for state change information to the Apple device.

The Display Remote commands export text as UTF-8 characters. Graphics cannot be exported. Note that Chapter count information can be retrieved only from the currently playing track.

Table 4-48 Display Remote lingo command summary

Command	ID	Data length	Protocol version	Requires authentication over UART
iPodAck	0x00	0x02	1.00	No
GetCurrentEQProfileIndex	0x01	0x00	1.00	No
RetCurrentEQProfileIndex	0x02	0x04	1.00	No
SetCurrentEQProfileIndex	0x03	0x05	1.00	No
GetNumEQProfiles	0x04	0x00	1.00	No
RetNumEQProfiles	0x05	0x04	1.00	No
GetIndexedEQProfileName	0x06	0x04	1.00	No
RetIndexedEQProfileName	0x07	0x*NN*	1.00	No
SetRemoteEventNotification	0x08	0x04	1.02	Yes
RemoteEventNotification	0x09	0x*NN*	1.02	Yes
GetRemoteEventStatus	0x0A	0x00	1.02	Yes
RetRemoteEventStatus	0x0B	0x04	1.02	Yes
GetiPodStateInfo	0x0C	0x01	1.02	Yes
RetiPodStateInfo	0x0D	0x*NN*	1.02	Yes
SetiPodStateInfo	0x0E	0x*NN*	1.02	Yes
GetPlayStatus	0x0F	0x00	1.02	Yes
RetPlayStatus	0x10	0x0D	1.02	Yes
SetCurrentPlayingTrack	0x11	0x04	1.02	Yes
GetIndexedPlayingTrackInfo	0x12	0x07	1.02	Yes
RetIndexedPlayingTrackInfo	0x13	0x*NN*	1.02	Yes
GetNumPlayingTracks	0x14	0x00	1.02	Yes
RetNumPlayingTracks	0x15	0x04	1.02	Yes
GetArtworkFormats	0x16	0x02	1.04	Yes
RetArtworkFormats	0x17	0x*NN*	1.04	Yes

Command	ID	Data length	Protocol version	Requires authentication over UART
GetTrackArtworkData	0x18	0x0C	1.04	Yes
RetTrackArtworkData	0x19	0x*NN*	1.04	Yes
GetPowerBatteryState	0x1A	0x00	1.02	No
RetPowerBatteryState	0x1B	0x02	1.02	No
GetSoundCheckState	0x1C	0x00	1.02	No
RetSoundCheckState	0x1D	0x01	1.02	No
SetSoundCheckState	0x1E	0x02	1.02	No
GetTrackArtworkTimes	0x1F	0x0C	1.04	Yes
RetTrackArtworkTimes	0x20	0x*NN*	1.04	Yes
CreateGeniusPlaylist	0x21	0x0C	1.05	Yes
IsGeniusAvailableForTrack	0x22	0x08	1.05	Yes
Reserved	0x23–0xFF	N/A	N/A	N/A

4.3.1 Command History of the Display Remote lingo

Table 4-49 (page 250) shows the history of command changes in the Display Remote lingo:

Table 4-49 Display Remote lingo command history

Lingo version	Command changes	Features
1.00	Add: 0x00–0x07	Apple device Equalizer Setting save/restore control
1.01	None	BugFix: Equalizer state not restored on Extended Interface mode exit
1.02	Add: 0x08–0x15, 0x1A–0x1E	Event notifications, Apple device state info, playback track info, sound check, power/battery support
1.03	None	BugFix: Fix intermittent UI hang when restoring on exit
1.04	Add: 0x16–0x19, 0x1F–0x20	Track artwork, lyrics, track time position in seconds
1.05	Add: 0x21-0x22	Video browsing support added to playback commands. Chapter information can be retrieved for all tracks in the Now Playing list, not just for the currently playing track. Added support for Genius playlists.

4.3.2 Transferring Album Art

The Display Remote lingo includes several commands that support the transfer of album artwork from an Apple device to an accessory:

- `GetArtworkFormats`
- `RetArtworkFormats`
- `GetTrackArtworkTimes`
- `RetTrackArtworkTimes`
- `GetTrackArtworkData`
- `RetTrackArtworkData`

All album art image encoding is RGB-565, which can be transferred in both big- and small-endian formats. Artwork retrieval takes place in the following steps:

1. Retrieve the number of formats available for artwork on an Apple device using `GetArtworkFormats`. It is not necessary to call `GetArtworkFormats` more than once per session; these values will be static while the accessory is attached to the Apple device. However, there are no guarantees about the number of formats and which ones are available on a particular model or firmware version. Each `formatID` in `RetArtworkFormats` specifies both a pixel encoding, such as RGB-565 little-endian, and the image dimensions. All formats are fixed-size.

2. When the accessory wants to retrieve the artwork for a given track, it calls `GetIndexedPlayingTrackInfo` with an `infoType` of 0x08. This returns the count of artwork available for each `formatID` associated with the track. It is possible that a track may not have artwork for a particular `formatID` or that the number of images will vary by `formatID`. A given size of album artwork may be available for only one track in the database.

3. To retrieve the list of images associated with a given track and `formatID`, the accessory calls `GetTrackArtworkTimes`. This command tells the Apple device to return the associated timestamp for each artwork. The timestamp indicates when the artwork should be displayed, expressed in milliseconds from the start of playback.

4. When the accessory wants to retrieve an individual piece of artwork, it sends `GetTrackArtworkData` to the Apple device. This requires the accessory to specify a track, a formatID, and the timestamp of the desired image. The Apple device returns the specified artwork and the accessory can display it whenever it chooses.

4.3.3 Command 0x00: iPodAck

Lingo: 0x03 — Origin: Apple device

The Apple device sends this command to acknowledge the receipt of a command from the accessory and return the command status. The command ID field indicates the accessory command for which the response is being sent. The command status indicates the result of the command (success or failure).

Table 4-50 `iPodAck` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command status; see Table 3-6 (page 125).
0x*NN*	1	The ID for the command being acknowledged.
Packet footer specified in "Command Packets" (page 109).		

4.3.4 Command 0x01: GetCurrentEQProfileIndex

Lingo: 0x03 — Origin: Accessory

The accessory requests the current Equalizer Profile setting index. In response, the Apple device sends the "Command 0x02: RetCurrentEQProfileIndex" (page 252) packet.

Table 4-51 `GetCurrentEQProfileIndex` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.3.5 Command 0x02: RetCurrentEQProfileIndex

Lingo: 0x03 — Origin: Apple device

The Apple device sends this command, returning the current Equalizer Profile setting index, in response to the "Command 0x01: GetCurrentEQProfileIndex" (page 252) packet sent by the accessory. An Equalizer Index of 0x0 indicates that the equalizer is disabled.

Table 4-52 `RetCurrentEQProfileIndex` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Current Equalizer Index
Packet footer specified in "Command Packets" (page 109).		

4.3.6 Command 0x03: SetCurrentEQProfileIndex

Lingo: 0x03 — Origin: Accessory

The accessory sets the current Equalizer Profile setting index and optionally restores the original Equalizer Setting on accessory detach. The valid Equalizer Index range can be determined by sending "Command 0x04: GetNumEQProfiles" (page 253). An Equalizer Index of 0x0 tells the Apple device that the equalizer should be disabled; a valid nonzero index enables the equalizer.

In response to this command, the Apple device returns an `iPodAck` packet with the status of this command.

Table 4-53 `SetCurrentEQProfileIndex` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Current Equalizer Index
0x*NN*	1	`bRestoreOnExit`. Specifies whether to restore the previous Equalizer Setting on accessory exit or detach. See the discussion below.
Packet footer specified in "Command Packets" (page 109).		

The `bRestoreOnExit` Boolean byte flag determines the behavior of the Apple device when the accessory is detached from the connector. A value of 0x0 (false) indicates that the original Equalizer Setting should be discarded. A nonzero (true) value indicates that the previous Equalizer Setting should be restored when the accessory is detached from the Apple device. Anytime the `SetCurrentEQProfileIndex` command is sent with `bRestoreOnExit` equal to false, the previous equalizer state is erased and lost. If `SetCurrentEQProfileIndex` is sent with `bRestoreOnExit` equal to true every time, the first `SetCurrentEQProfileIndex` command saves the original equalizer state; subsequent commands do not change the original saved equalizer state. On accessory detach, the original saved equalizer state is restored.

4.3.7 Command 0x04: GetNumEQProfiles

Lingo: 0x03 — Origin: Accessory

The accessory requests the number of Apple device Equalizer Profile settings. In response, the Apple device sends the "Command 0x05: RetNumEQProfiles" (page 254) packet.

Table 4-54 `GetNumEQProfiles` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.3.8 Command 0x05: RetNumEQProfiles

Lingo: 0x03 — Origin: Apple device

The Apple device returns the number of Equalizer Profiles in it. It sends this command in response to the "Command 0x04: GetNumEQProfiles" (page 253) packet sent by the accessory. The valid profile index range for Apple device equalizer commands accepting a profile index is 0x0 to `profileCount-1`.

Table 4-55 `RetNumEQProfiles` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	`profileCount`. The Equalizer Profile count .
Packet footer specified in "Command Packets" (page 109).		

4.3.9 Command 0x06: GetIndexedEQProfileName

Lingo: 0x03 — Origin: Accessory

The accessory requests the Apple device Equalizer Profile setting name for a given Equalizer Profile index. In response, the Apple device sends the "Command 0x07: RetIndexedEQProfileName" (page 254) packet. The valid profile index range can be obtained by sending "Command 0x04: GetNumEQProfiles" (page 253).

Table 4-56 `GetIndexedEQProfileName` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Equalizer profile index
Packet footer specified in "Command Packets" (page 109).		

4.3.10 Command 0x07: RetIndexedEQProfileName

Lingo: 0x03 — Origin: Apple device

The Apple device returns its Equalizer Profile setting name for the specified Equalizer Profile index in response to "Command 0x06: GetIndexedEQProfileName" (page 254). The Equalizer Profile name is returned as a variable-length, null-terminated UTF-8 character array.

Note: The UTF-8 Equalizer Profile name string is not limited to 255 characters. It may be sent in either small or large packet format. The following table shows the small packet format.

Table 4-57 `RetIndexedEQProfileName` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	*NN*	The Equalizer Profile name, as a null-terminated UTF-8 character array.
Packet footer specified in "Command Packets" (page 109).		

4.3.11 Command 0x08: SetRemoteEventNotification

Lingo: 0x03 — Origin: Accessory

The accessory requests that the Apple device enable asynchronous remote event notification for specific Apple device events. Notification for each event can be enabled by setting the associated bit in the remote event bitmask (`remEventMask`). By default, all event notifications are disabled and must be explicitly enabled using this command. In response, the Apple device sends an `iPodAck` command indicating the command completion status. A remote event bitmask of 0x0 disables all remote event status notifications. On accessory detach, event notification is reset to the default disabled state.

Table 4-58 `SetRemoteEventNotification` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	`remEventMask` . See Table 4-59 (page 255) for a list of the events for which you can enable notification.
Packet footer specified in "Command Packets" (page 109).		

Enable notifications for the events listed in Table 4-59 (page 255) by setting the bit for each event in the remote event bitmask. A value of 1 enables the notification of the Apple device state change for that event and a value of 0 disables the notification.

Table 4-59 Apple device events

Bit number	Remote Event
0	Track time position in milliseconds
1	Track playback index
2	Chapter index
3	Play status (play, pause, stop, FF, and RW)

Bit number	Remote Event
4	Mute/UI Volume (see Notes below)
5	Power/battery
6	Equalizer setting
7	Shuffle setting
8	Repeat setting
9	Date and time setting
10	Alarm setting **Deprecated; do not use**
11	Backlight level
12	Hold switch state
13	Sound check state
14	Audiobook speed
15	Track time position in seconds
16	Mute/UI/Absolute Volume (see Notes below)
17	Track capabilities
18	Playback engine contents
31:19	Reserved

Notes to table: When an accessory has identified itself using IDPS, it must do the following before setting bit 4 or bit 16 in the remote event bitmask of `RemoteEventNotification` to obtain notifications of volume change events:

- It must set bit 00 and bit 11 in its `AccessoryCapsToken` value to enable analog line out and to check its volume; see Table 3-71 (page 164).
- It must register a `SetiPodPreferenceToken` of Class 0x03 and value 0x01 to enable line out; see Table 3-57 (page 152).

An accessory should set bit 4 or bit 16 only if it can respond appropriately to the corresponding notifications.

4.3.12 Command 0x09: RemoteEventNotification

Lingo: 0x03 — Origin: Apple device

The Apple device sends this command asynchronously whenever an enabled event change has occurred. Use "Command 0x08: SetRemoteEventNotification" (page 255) to control which events are enabled. The notification packet formats are described in more detail below. Notifications for enabled events are sent every 500 ms, with the exception of volume change notifications, which are sent every 100 ms.

Note: Notifications are sent only when the Apple device is in Power On mode; none are sent when the Apple device is in Sleep or Hibernate mode.

This is the command packet for the `RemoteEventNotifcation` command. See Table 4-61 (page 257) to interpret the `eventNum` and `eventData` fields.

Table 4-60 `RemoteEventNotification` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`eventNum`. A number indicating the type of event.
0x*NN*	*NN*	`eventData`. Additional information about the event. This field is variable length; see Table 4-61 (page 257).
Packet footer specified in "Command Packets" (page 109).		

Table 4-61 (page 257) shows the expected payload and length for each type of event notification. For example, if an accessory receives an event notification for event 0x02 (Chapter Info) then the payload is 8 bytes long and contains the track index, the chapter count, and the chapter index. If the chapter count bytes are zero, the currently playing track does not have chapters.

Table 4-61 Event notification data

Event number	Event	Event data	Data length in bytes
0x00	Track position	The new track position, in milliseconds. ■ Byte 0: Track Position (bits 31:24) ■ Byte 1: Track Position (bits 23:16) ■ Byte 2: Track Position (bits 15:8) ■ Byte 3: Track Position (bits 7:0)	0x04
0x01	Track index	The index of the currently playing track. ■ Byte 0: Track Index (bits 31:24) ■ Byte 1: Track Index (bits 23:16) ■ Byte 2: Track Index (bits 15:8) ■ Byte 3: Track Index (bits 7:0)	0x04

Event number	Event	Event data	Data length in bytes
0x02	Chapter information	Chapter information, including the track index, chapter count, and chapter index. ■ Bytes 0–3 specify the currently playing track index, as follows: Byte 0: Track Index (bits 31:24) Byte 1: Track Index (bits 23:16) Byte 2: Track Index (bits 15:8) Byte 3: Track Index (bits 7:0) ■ Bytes 4-5 specify the chapter count of the track, as follows. A value of 0x0000 indicates that the track does not have chapters. Byte 4: Chapter Count (bits 15:8) Byte 5: Chapter Count (bits 7:0) ■ Bytes 6-7 specify the chapter index, as follows. A value of 0xFFFF indicates that the track does not have chapters. Byte 6: Chapter Index (bits 15:8) Byte 7: Chapter Index (bits 7:0)	0x08
0x03	Play status	The current play status of the Apple device; that is, whether it is playing, paused, stopped, fast forwarding or rewinding. Possible values are described in Table 4-62 (page 262). ■ Byte 0: Play Status (bits 7:0)	0x01
0x04	Mute/UI Volume	The current state of the mute setting and UI volume information. ■ Byte 0: Mute State (bits 7:0) A value of 0 indicates that mute is off; a value of 1 indicates that mute is on. ■ Byte 1: UI Volume Level (bits 7:0) A value between 0 and 255, with 0 indicating minimum volume and 255 indicating maximum volume. Note that if the Mute State value is true (mute is on), the UI volume level field is not valid and is returned as 0.	0x02

Event number	Event	Event data	Data length in bytes
0x05	Power/battery	Information about the power and battery status. ■ Byte 0: Power State (bits 7:0) A value indicating the current power source and its state. See Table 4-65 (page 263) for possible values. ■ Byte 1: Battery Level (bits 7:0) Specifies the current battery level. A value from 0 to 255, with 0 indicating a fully discharged battery and 255 indicating a battery that is fully charged. If an external power status is returned, the battery level is invalid and is returned as 0.	0x02
0x06	Equalizer state	The current Equalizer Setting index. ■ Byte 0: Equalizer index (bits 31:24) ■ Byte 1: Equalizer index (bits 23:16) ■ Byte 2: Equalizer index (bits 15:8) ■ Byte 3: Equalizer index (bits 7:0)	0x04
0x07	Shuffle	The state of the shuffle setting. See Table 4-63 (page 262) for a list of possible values. ■ Byte 0: Shuffle State (bits 7:0)	0x01
0x08	Repeat	The state of the repeat setting. See Table 4-64 (page 262) for a list of possible values. ■ Byte 0: Repeat State (bits 7:0)	0x01

Event number	Event	Event data	Data length in bytes
0x09	Date/time	The current date and time. ■ Bytes 0–1 specify the current year. A value of 2005 represents the year 2005 A.D. Byte 0: Year (bits 15:8) Byte 1: Year (bits 7:0) ■ Byte 2: Month (bits 7:0) A value between 1 and 12, where 1 = January and 12 = December. ■ Byte 3: Day of the month (bits 7:0) A value between 1 and 31. ■ Byte 4: Hour (bits 7:0) A value between 0 and 23, where 0 = 12:00 a.m. and 23 = 11:00 p.m. ■ Byte 5: Minute (bits 7:0) A value between 0 and 59.	0x06
0x0A	Alarm	**Deprecated; do not use**	0x03
0x0B	Backlight	The current backlight level. A value between 0 and 255, where 0 indicates the backlight is at minimum brightness and 255 indicates that the backlight is at full intensity. ■ Byte 0: Backlight Level (bits 7:0)	0x01
0x0C	Hold switch	The current state of the hold switch. A value of 0 means the hold switch is off. A value of 1 indicates it is on. ■ Byte 0: Hold Switch State (bits 7:0)	0x01
0x0D	Sound check	The state of the sound check setting. A value of 0 means that sound check is off; a value of 1 indicates it is on. ■ Byte 0: Sound Check State (bits 7:0)	0x01
0x0E	Audiobook	The audiobook playback speed setting. See Table 4-66 (page 263) for a list of possible values. ■ Byte 0: Audiobook Playback Speed Setting (bits 7:0)	0x01
0x0F	Track position in seconds	The new track time position, in seconds. ■ Byte 0: Track Position (bits 15:8) ■ Byte 1: Track Position (bits 7:0)	0x02

Event number	Event	Event data	Data length in bytes
0x10	Mute/UI/Absolute Volume	The current state of the mute setting, UI volume, and absolute volume. ■ Byte 0: Mute state (bits 7:0). A value of 0 indicates that muting is off; a value of 1 indicates that muting is on. ■ Byte 1: UI volume level (bits 7:0). A value between 0 and 255, normalized to volume limit settings. 0 indicates minimum volume and 255 indicates maximum volume. ■ Byte 2: Absolute volume level (bits 7:0). A value between 0 and 255, not normalized. 0 indicates minimum absolute volume and 255 indicates maximum absolute volume. If muting is on (byte 0 = 0x00), the absolute volume level is not valid and is returned as 0.	0x03
0x11	Track capabilities	Track capabilities bits: ■ Bit 0: If equal to 1, the track is an audiobook. ■ Bit 1: If equal to 1, the track has chapters. ■ Bit 2: Set to 1 if album artwork is available, 0 otherwise. ■ Bit 3: If equal to 1, the track has song lyrics. ■ Bit 4: If equal to 1, the track is a podcast episode. ■ Bit 5: Track has release date. ■ Bit 6: Track has description. ■ Bit 7: Track contains video (a video podcast, music video, movie, or TV show). ■ Bit 8: Track is currently queued to play as a video. ■ Bit 9-12: Reserved. ■ Bit 13: Track is capable of generating a Genius playlist. ■ Bit 14: If equal to 1, the track is an iTunesU episode. ■ Bit 31:15: Reserved.	0x04
0x12	Playback engine contents changed	Bytes 0-3: number of tracks in new playlist.	0x04
0x13–0xFF	Reserved	N/A	N/A

The battery and volume UI levels can both range from 0 (lowest) to 255 (highest). The absolute volume level ranges between 0 and the Apple device's Volume Limit setting. The UI volume is scaled to cover the Absolute volume range; thus, setting the UI volume to 255 will result in the Absolute volume being set to the Apple device's Volume Limit setting. The granularity of minimum to maximum range steps varies between Apple devices.

Table 4-62 (page 262) lists the possible values for the data associated with a Play Status event and their meanings.

Table 4-62 Play status values

Value	Meaning
0x00	Playback stopped
0x01	Playing (for "Command 0x0E: SetiPodStateInfo" (page 266), start or resume playback)
0x02	Playback paused
0x03	Fast forward (FF)
0x04	Fast rewind (REW)
0x05	End fast forward or rewind mode
0x06–0xFF	Reserved

Note: With the iPhone, incoming phone calls can pause audio playback at any time. The accessory should enable notifications for this event and act accordingly (for example, by changing an icon). It should not try to cancel the pause.

Table 4-63 (page 262) lists the possible values for the data associated with a Shuffle event.

Table 4-63 Shuffle state

Value	Meaning
0x00	Shuffle off
0x01	Shuffle tracks and songs
0x02	Shuffle albums
0x03–0xFF	Reserved

Table 4-64 (page 262) lists the possible values for the data associated with a Repeat event.

Table 4-64 Repeat state

Value	Meaning
0x00	Repeat off
0x01	Repeat one track or song
0x02	Repeat all tracks
0x03–0xFF	Reserved

Table 4-65 (page 263) lists the possible values for the data associated with a Power/Battery event and their meanings.

Table 4-65 Power and battery state

Value	Meaning
0x00	Internal battery power, low power (< 30%)
0x01	Internal battery power
0x02	External power, battery pack, no charging
0x03	External power, no charging
0x04	External power, battery charging
0x05	External power, battery charged
0x06–0xFF	Reserved

Table 4-66 (page 263) lists the possible values for the data associated with an Audiobook event and their meanings.

Table 4-66 Audiobook playback speeds

Value	Meaning
0xFF	Slower (–1)
0x00	Normal
0x01	Faster (+1)
0x02–0xFE	Reserved

4.3.13 Command 0x0A: GetRemoteEventStatus

Lingo: 0x03 — Origin: Accessory

The accessory requests the status of state information that has changed on the Apple device. In response, the Apple device sends "Command 0x0B: RetRemoteEventStatus" (page 264), containing a bitmask of event states that changed since the last `GetRemoteEventStatus` command and clears all the remote event status bits. This command may be used to poll the Apple device for certain event changes without enabling asynchronous remote event notification.

Table 4-67 `GetRemoteEventStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		

Value	Bytes	Parameter
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.3.14 Command 0x0B: RetRemoteEventStatus

Lingo: 0x03 — Origin: Apple device

The Apple device sends this command in response to "Command 0x0A: GetRemoteEventStatus" (page 263).

Table 4-68 `RetRemoteEventStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	`remEventStatus` . The event status that has changed. See Table 4-59 (page 255) for a list of possible events.
Packet footer specified in "Command Packets" (page 109).		

The bits in Table 4-59 (page 255) represent the events whose status has changed on the Apple device since the last `GetRemoteEventStatus` command was received. For example, if the returned `remEventStatus` field has bits 2 and 7 set, then the chapter index and shuffle states of the Apple device have changed since the last `GetRemoteEventStatus` command was sent by the accessory. Accessories can use "Command 0x0C: GetiPodStateInfo" (page 264) to get the updated state information for those events.

4.3.15 Command 0x0C: GetiPodStateInfo

Lingo: 0x03 — Origin: Accessory

The accessory obtains Apple device state information. The information type (`infoType`) field specifies the type of information to get. In response, the Apple device sends "Command 0x0D: RetiPodStateInfo" (page 265) with the requested state information.

Table 4-69 `GetiPodStateInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`infoType`: The type of state information for which to query the Apple device. See Table 4-74 (page 266).
Packet footer specified in "Command Packets" (page 109).		

Table 4-74 (page 266) lists the different types of information for which you can query the Apple device.

Note: An accessory should request `infoType` values of 0x04 (Mute/UI volume) or 0x10 (Mute/UI/Absolute volume) only if it can respond appropriately to the corresponding information.

For example, to retrieve the Apple device's Equalizer Setting, an accessory could send the command shown in Table 4-70 (page 265).

Table 4-70 `GetiPodStateInfo` packet to retrieve the Apple device Equalizer Setting

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x06	1	`infoType` = 0x06 (Equalizer setting)
Packet footer specified in "Command Packets" (page 109).		

4.3.16 Command 0x0D: RetiPodStateInfo

Lingo: 0x03 — Origin: Apple device

The Apple device sends this command in response to "Command 0x0C: GetiPodStateInfo" (page 264). The format of the returned state information depends on the type of information.

Table 4-71 `RetiPodStateInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`infoType`. The type of Apple device state information returned; see Table 4-61 (page 257).
0x*NN*	*NN*	`infoData`. The Apple device state information; see Table 4-61 (page 257).
Packet footer specified in "Command Packets" (page 109).		

For example, an accessory requesting the Chapter Info of the currently playing track would receive a `RetiPodStateInfo` command packet similar to this (assuming a track index of 10, a chapter count of 8 and a current chapter index of 3):

Table 4-72 `RetiPodStateInfo` packet for requesting chapter information

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x02	1	`infoType` = 0x02 (Chapter Information)
0x00	4	`infoData` = Track Index
0x00	2	`infoData` = Chapter Count

Value	Bytes	Parameter
0x00	2	`infoData` = Chapter Index
Packet footer specified in "Command Packets" (page 109).		

4.3.17 Command 0x0E: SetiPodStateInfo

Lingo: 0x03 — Origin: Accessory

Sets the Apple device state. The information type (`infoType`) field specifies the type of information to update. In response, the Apple device sends an `iPodAck` command with the results of the operation. Some commands include a `bRestoreOnExit` parameter that optionally allows the original Apple device setting to be restored on exit.

Table 4-73 `SetiPodStateInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`infoType`. The type of Apple device state information to set.
0x*NN*	1	`infoData` (variable length). The data for the Apple device state information to set.
Packet footer specified in "Command Packets" (page 109).		

Table 4-74 (page 266) lists the possible values of the `infoType` field and the corresponding data in the `infoData` field.

Note: Information type 0x09 (date/time) cannot be set on an iOS device.

Table 4-74 Apple device state data

Information type		Information data	
Value	**Name**	**Description**	**Length (bytes)**
0x00	Track position	The new position of the track, in milliseconds. ■ Byte 0: Track Position (bits 31:24) ■ Byte 1: Track Position (bits 23:16) ■ Byte 2: Track Position (bits 15:8) ■ Byte 3: Track Position (bits 7:0)	0x04

Information type		Information data	
Value	**Name**	**Description**	**Length (bytes)**
0x01	Track index	The index of the track to play. ■ Byte 0: Track Index (bits 31:24) ■ Byte 1: Track Index (bits 23:16) ■ Byte 2: Track Index (bits 15:8) ■ Byte 3: Track Index (bits 7:0)	0x04
0x02	Chapter index	The new chapter index. ■ Byte 0: Chapter Index (bits 15:8) ■ Byte 1: Chapter Index (bits 7:0)	0x02
0x03	Play status	The play status of the Apple device (play, pause, stop, FF or REW). See Table 4-62 (page 262) for a list of possible values. Byte 0: Play Status (bits 7:0)	0x01
0x04	Mute/UI volume	The new mute setting or UI volume level. ■ Byte 0: Mute State (bits 7:0) A value of 0x00 turns mute off; a value of 0x01 turns on mute. ■ Byte 1: UI volume Level (bits 7:0) A value between 0 and 255, with 0 indicating minimum UI volume and 255 indicating maximum UI volume. ■ Byte 2: `bRestoreOnExit` (bits 7:0) See Table 4-75 (page 270). If the mute state is 0x01, the UI volume level field is ignored.	0x03
0x05	Power/battery	Reserved: Power and battery state cannot be set.	N/A
0x06	Equalizer state	The new Equalizer Setting. ■ Byte 0: Equalizer index (bits 31:24) ■ Byte 1: Equalizer index (bits 23:16) ■ Byte 2: Equalizer index (bits 15:8) ■ Byte 3: Equalizer index (bits 7:0) ■ Byte 4: `bRestoreOnExit` (bits 7:0) See Table 4-75 (page 270).	0x05

Information type		Information data	
Value	**Name**	**Description**	**Length (bytes)**
0x07	Shuffle	The new state of the shuffle setting. ■ Byte 0: Shuffle State (bits 7:0) See Table 4-63 (page 262) for a list of possible values. ■ Byte 1: `bRestoreOnExit` (bits 7:0) See Table 4-75 (page 270).	0x02
0x08	Repeat	The new state of the repeat setting ■ Byte 0: Repeat State (bits 7:0) See Table 4-64 (page 262) for a list of possible values. ■ Byte 1: `bRestoreOnExit` (bits 7:0) See Table 4-75 (page 270).	0x02
0x09	Date/time	The new date and time. ■ Bytes 0–1 specify the current year. A value of 2005 represents the year 2005 A.D. Byte 0: Year (bits 15:8) Byte 1: Year (bits 7:0) ■ Byte 2: Month (bits 7:0) A value between 1 and 12, where 1 = January and 12 = December. ■ Byte 3: Day of the month (bits 7:0) A value between 1 and 31. ■ Byte 4: Hour (bits 7:0) A value between 0 and 23, where 0 = 12:00 a.m. and 23 = 11:00 p.m. ■ Byte 5: Minute (bits 7:0) A value between 0 and 59.	0x06
0x0A	Alarm	**Deprecated; do not use**	0x04
0x0B	Backlight	The new level of the backlight. ■ Byte 0: Backlight Level (bits 7:0) The backlight level over a range of 0 (minimum brightness) to 255 (full intensity). ■ Byte 1: Reserved; set to 1 (bits 7:0)	0x02
0x0C	Hold switch	Reserved. The state of the Hold switch cannot be set.	N/A

Information type		Information data	
Value	**Name**	**Description**	**Length (bytes)**
0x0D	Sound check	The new sound check state. ■ Byte 0: Sound Check State (bits 7:0) A value of 0x00 turns sound check off; a value of 0x01 turns it on. ■ Byte 1: `bRestoreOnExit` (bits 7:0) See Table 4-75 (page 270).	0x02
0x0E	Audiobook speed	The audiobook playback speed. ■ Byte 0: Audiobook Playback Speed Setting (bits 7:0) See Table 4-66 (page 263) for a list of possible values. ■ Byte 1: `bRestoreOnExit` (bits 7:0) See Table 4-75 (page 270).	0x02
0x0F	Track position in seconds	The current track time position, in seconds. ■ Byte 0: Track Time Position (bits 15:8) ■ Byte 1: Track Time Position (bits 7:0)	0x02
0x10	Mute/UI/Absolute volume	The current state of the mute setting, UI volume, and Absolute volume. ■ Byte 0: Mute state (bits 7:0). A value of 0 indicates that muting is off; a value of 1 indicates that muting is on. ■ Byte 1: UI volume level (bits 7:0). A value between 0 and 255, normalized to UI volume limit settings. 0 indicates minimum UI volume and 255 indicates maximum UI volume. If the accessory sets this byte to 0, the Apple device uses the Absolute volume setting. ■ Byte 2: Absolute volume level (bits 7:0). A value between 0 and 255, not normalized. 0 indicates minimum Absolute volume and 255 indicates maximum Absolute volume. If muting is on (byte 0 = 0x00), the Absolute volume level is not valid and is returned as 0. ■ Byte 3: `bRestoreOnExit` (bits 7:0). See Table 4-75 (page 270) for a list of values.	0x04
0x11–0xFF	Reserved	N/A	N/A

Table 4-75 (page 270) lists the possible values for the `bRestoreOnExit` parameter.

Table 4-75 Restore-on-exit values

Value	Meaning
0x00	Do not save the original state.
0x01	Save the original state and restore it on exit.

For example, an accessory could send the command shown in Table 4-76 (page 270) to set the currently playing track on the Apple device to track 1000.

Table 4-76 `SetiPodStateInfo` packet for setting the current track

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x01	1	`infoType` = 0x01 (Track Index)
0x00	4	`infoData` = Track Index. Sets the playback track index to 1000.
Packet footer specified in "Command Packets" (page 109).		

Note: `SetiPodStateInfo` commands that use information types that have data fields larger than one byte must be sure to send the data in big-endian format. See the example in Table 4-76 (page 270).

4.3.18 Command 0x0F: GetPlayStatus

Lingo: 0x03 — Origin: Accessory

The accessory requests the current Apple device play status information. In response, the Apple device sends "Command 0x10: RetPlayStatus" (page 270) with the current play state, track index, track position, and track length.

Table 4-77 `GetPlayStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.3.19 Command 0x10: RetPlayStatus

Lingo: 0x03 — Origin: Apple device

The Apple device sends this command in response to "Command 0x0F: GetPlayStatus" (page 270) and returns the current Apple device play status information. If the Apple device is in a playing or paused state, the track index (`trackIndex`), track length (`trackTotMs`), and track position (`trackPosMs`) fields are valid. Otherwise, they should be ignored.

Table 4-78 `RetPlayStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`playState`. The Apple device playback engine state. See Table 4-62 (page 262) for a list of possible values. If the value of this field is 0x00, all of the subsequent track fields are invalid. The value 0x05 is reserved.
0x*NN*	4	`trackIndex`. Specifies the index of the currently playing track.
0x*NN*	4	`trackTotMs`. Specifies the total length of the track, in milliseconds.
0x*NN*	4	`trackPosMs`. The current position of the track, in milliseconds.
Packet footer specified in "Command Packets" (page 109).		

4.3.20 Command 0x11: SetCurrentPlayingTrack

Lingo: 0x03 — Origin: Accessory

The accessory sets the Apple device's currently playing track to the track at the specified index. The total number of playing tracks can be obtained by sending "Command 0x14: GetNumPlayingTracks" (page 274). The playing track index is zero based, so the valid range is from 0x0 to `numPlayTracks`–1 (one less than the total count).

Table 4-79 `SetCurrentPlayingTrack` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	`trackIndex`. The track index to begin playing.
Packet footer specified in "Command Packets" (page 109).		

4.3.21 Command 0x12: GetIndexedPlayingTrackInfo

Lingo: 0x03 — Origin: Accessory

The accessory requests track information for the specified playing track index. The `infoType` field specifies the type of information to be returned, such as track title, artist name, album name, track genre, and track chapter information. In response, the Apple device sends "Command 0x13: RetIndexedPlayingTrackInfo" (page 272) with the requested track information.

Table 4-80 `GetIndexedPlayingTrackInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`infoType`. The type of track information to retrieve. See Table 4-82 (page 273) for a list of the available types of information.
0x*NN*	4	`trackIndex` . The index of the track for which to retrieve information.
0x*NN*	2	`chapIndex` . The index of the chapter for which to retrieve information. This field is valid only when `infoType` is chapter information (0x01) and the track at `trackIndex` has chapters. Set the `chapIndex` fields to 0x00 if `infoType` is not chapter information and `trackIndex` does not have chapters.
Packet footer specified in "Command Packets" (page 109).		

4.3.22 Command 0x13: RetIndexedPlayingTrackInfo

Lingo: 0x03 — Origin: Apple device

The Apple device sends this command in response to "Command 0x12: GetIndexedPlayingTrackInfo" (page 271). It returns the requested type of information and data for the specified playing track. Data returned as strings are encoded as null-terminated UTF-8 character arrays. If the track information string does not exist, an empty string is returned. The accessory must not assume that such an empty string will be null-terminated.

Table 4-81 `RetIndexedPlayingTrackInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`infoType`. The type of track information being returned. See Table 4-82 (page 273) for possible values.
0x*NN*	*NN*	`infoData` The track information data. The Apple device sends a command packet with a `length` field that is not larger than the accessory's maximum payload size (see "Payload Length Field" (page 110)).
Packet footer specified in "Command Packets" (page 109).		

Table 4-82 (page 273) shows the data returned for each type of track information.

Table 4-82 Track information data

Info type	Data type	Track information data	Data length
0x00	Track caps/info	Track capabilities and information. This contains 3 fields: ■ Bytes 0–3: Track Caps bitfields. Specifies the capabilities of the track. These bits have the following meanings: Bit 0: If equal to 1, the track is an audiobook. Bit 1: If equal to 1, the track has chapters. Bit 2: Set to 1 if album artwork is available, 0 otherwise. Bit 3: If equal to 1, the track has song lyrics. Bit 6:4: Reserved. Bit 7: Track contains video (a video podcast, music video, movie, or TV show). Bit 8: Track is currently queued to play as a video. Bit 12:9: Reserved. Bit 13: Track is capable of generating a Genius playlist. Bit 14: Track is an iTunesU episode. Bit 31:15: Reserved. ■ Bytes 4–7: TrackTotMs. The total length of the track in milliseconds. ■ Bytes 8–9: ChapCount. If the track has chapters, the chapter count.	10
0x01	Chapter time/name	Chapter time and name, having two fields: ■ Bytes 0–3: Chapter Time The chapter offset time in milliseconds. ■ (variable length): Chapter Name The chapter name, as a UTF-8 string.	4 + Variable
0x02	Artist name	The artist name, as a UTF-8 string.	Variable
0x03	Album name	The album name, as a UTF-8 string.	Variable
0x04	Genre name	The genre name, as a UTF-8 string.	Variable
0x05	Track title	The title of the track, as a UTF-8 string.	Variable
0x06	Composer name	The composer name, as a UTF-8 string.	Variable

Info type	Data type	Track information data	Data length
0x07	Lyrics	Track lyrics data, consisting of 3 fields: ■ Byte 0: Packet information bits. If set, these bits have the following meanings: Bit 0: If equal to 1, indicates that this is one of multiple packets. Bit 1: If equal to 1, this is the last packet (applicable only if bit 0 is equal to 1). Bits 7:2: Reserved; set to 0. ■ Bytes 1–2: Packet index (16-bit big-endian format) ■ Bytes 3–*NN*: Track lyrics as a UTF-8 string (see **Note** below).	Variable
0x08	Artwork count	Artwork count data. The artwork count is a sequence of 4-byte records; each record consists of a 2-byte `formatID` value followed by a 2-byte count of images in that format for this track. If the track contains no format IDs, this info type will return only 1 byte containing 0x08 and no records. For information about `formatID` values, see "Command 0x17: RetArtworkFormats" (page 275).	Variable
0x09–0xFF	Reserved	N/A	N/A

Note: Track lyrics are formatted as a single null-terminated UTF8 string. If the lyrics string is too long to be carried within a single packet, then the string is broken into sections and carried within separate packets with distinct values for each section index. The last lyrics packet section contains the null terminator character for the full track lyrics string. Lyric sections before the last section do not include null terminators, and individual sections may not necessarily be valid UTF8 strings. Accessories must use the packet payload length to determine the length of each string section and assemble the full lyrics string by concatenating the individual substrings in order.

If `GetIndexedPlayingTrackInfo` specified a track not currently playing, a null string is returned. Lyrics for a track being played may take up to 5 seconds to return. An accessory may try to get the current lyrics every 0.5 seconds until either 5 seconds has elapsed or the track lyrics string has been returned.

4.3.23 Command 0x14: GetNumPlayingTracks

Lingo: 0x03 — Origin: Accessory

The accessory requests the total number of tracks playing in the Apple device playback engine. The count can be used to select a different playing track or obtain information for a specific track. In response, the Apple device sends "Command 0x15: RetNumPlayingTracks" (page 275).

Table 4-83 `GetNumPlayingTracks` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.3.24 Command 0x15: RetNumPlayingTracks

Lingo: 0x03 — Origin: Apple device

The Apple device sends this command in response to "Command 0x14: GetNumPlayingTracks" (page 274) received from the accessory. It returns the total number of tracks queued in the playback engine.

Table 4-84 `RetNumPlayingTracks` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	`numPlayTracks`. The total number of queued playing tracks.
Packet footer specified in "Command Packets" (page 109).		

4.3.25 Command 0x16: GetArtworkFormats

Lingo: 0x03 — Origin: Accessory

The accessory sends this command to obtain the list of supported artwork formats on the Apple device. No parameters are sent. See "Transferring Album Art" (page 251).

Table 4-85 `GetArtworkFormats` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.3.26 Command 0x17: RetArtworkFormats

Lingo: 0x03 — Origin: Apple device

The Apple device sends this command to the accessory in response to "Command 0x16: GetArtworkFormats" (page 275). Each format is described in a 7-byte record (`formatID:2`, `pixelFormat:1`, `width:2`, `height:2`). The `formatID` is used when sending `GetTrackArtworkTimes`. The accessory may return zero records if the Apple device does not contain any artwork. See "Transferring Album Art" (page 251).

Table 4-86 `RetArtworkFormats` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	2	`formatID` Apple device-assigned value for this format
0x*NN*	1	`pixelFormat`. See Table 4-87 (page 276).
0x*NN*	2	`imageWidth`. Number of pixels wide for each image.
0x*NN*	2	`imageHeight`. Number of pixels high for each image.
0x*NN*	*NN*	Previous 4 parameter fields may be repeated *NN* times.
Packet footer specified in "Command Packets" (page 109).		

Table 4-87 Display pixel format codes

Display pixel format	Code
Reserved	0x00
Monochrome, 2 bits per pixel	0x01
RGB 565 color, little-endian, 16 bpp	0x02
RGB 565 color, big-endian, 16 bpp	0x03
Reserved	0x04–0xFF

4.3.27 Command 0x18: GetTrackArtworkData

Lingo: 0x03 — Origin: Accessory

Note: This command has been superseded by the Extended Interface lingo "Command 0x004E: GetArtworkData" (page 459), which provides better functionality.

The accessory sends this command to the Apple device to request the artwork image bitmap data for a given `trackIndex`, `formatID`, and `artworkIndex`. See "Transferring Album Art" (page 251).

Table 4-88 `GetTrackArtworkData` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	`trackIndex`.
0x*NN*	2	`formatID`
0x*NN*	4	Time offset in milliseconds
Packet footer specified in "Command Packets" (page 109).		

4.3.28 Command 0x19: RetTrackArtworkData

Lingo: 0x03 — Origin: Apple device

Note: This command has been superseded by the Extended Interface lingo "Command 0x004F: RetArtworkData" (page 460), which provides better functionality.

The Apple device sends this command in response to a "Command 0x18: GetTrackArtworkData" (page 276) command received from an accessory. Multiple `RetTrackArtworkData` commands may be necessary to transfer all the data because it will be too much to fit into a single packet. See "Transferring Album Art" (page 251).

This command returns the overall image width and height in pixels, the image row size in bytes, and the inset rectangle that contains the actual artwork content. The inset rectangle coordinates consist of two *x*,*y* pairs. Each *x* or *y* value is 2 bytes, so the total size of the inset rectangle coordinate set is 8 bytes.

Note: Track artwork may have its own border. In this case, the inset rectangle encloses only the actual artwork inside the border, not the whole image.

The total image size in bytes is given by multiplying the row size by the image height. Padding may be inserted at the top, bottom, left, or right of the artwork to fit it to the Apple device screen.

Table 4-89 `RetTrackArtworkData` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	2	Descriptor packet index. These fields uniquely identify each packet in the `RetTrackArtworkData` transaction. The first packet is the descriptor packet, which always starts with an index of 0x0000.
0x*NN*	1	Display pixel format code. See Table 4-87 (page 276).
0x*NN*	2	Image width in pixels

Value	Bytes	Parameter
0x*NN*	2	Image height in pixels
0x*NN*	2	Inset rectangle, top-left point, *x* value
0x*NN*	2	Inset rectangle, top-left point, *y* value
0x*NN*	2	Inset rectangle, bottom-right point, *x* value
0x*NN*	2	Inset rectangle, bottom-right point, *y* value
0x*NN*	4	Row size in bytes
0x*NN*...	*NN*	Image pixel data (variable length)
Packet footer specified in "Command Packets" (page 109).		

Note: In subsequent packets in the sequence (packets with a descriptor packet index greater than 0x0000), the Display pixel format code, Image width and height, Inset rectangle coordinates, and Row size fields are omitted. Only the Descriptor packet index and Image pixel data are included in the packet parameters.

4.3.29 Command 0x1A: GetPowerBatteryState

Lingo: 0x03 — Origin: Accessory

The accessory sends this command to obtain the power and battery level state of the Apple device. In response, the Apple device sends "Command 0x1B: RetPowerBatteryState" (page 278) with the power and battery information.

Table 4-90 `GetPowerBatteryState` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.3.30 Command 0x1B: RetPowerBatteryState

Lingo: 0x03 — Origin: Apple device

The Apple device sends this command in response to "Command 0x1A: GetPowerBatteryState" (page 278), returning the current Apple device power state and battery level.

Note: If an external power status (indicated by the value of the `powerStat` field) is returned, the battery level is invalid and is 0 on return.

Table 4-91 `RetPowerBatteryState` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`powerStat`. The battery power state. See Table 4-65 (page 263).
0x*NN*	1	`battLevel`. The battery power level. This is a value between 00 (the battery is fully discharged, no power remains) and 255 (the battery is fully charged). This field is valid only if `powerStat` is Internal battery (0x00 or 0x01)
Packet footer specified in "Command Packets" (page 109).		

4.3.31 Command 0x1C: GetSoundCheckState

Lingo: 0x03 — Origin: Accessory

The accessory requests the Apple device's current sound check setting. When enabled, sound check adjusts track playback volume to the same level. In response, the Apple device sends "Command 0x1D: RetSoundCheckState" (page 279) with the current sound check state.

Table 4-92 `GetSoundCheckState` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.3.32 Command 0x1D: RetSoundCheckState

Lingo: 0x03 — Origin: Apple device

The Apple device sends this command in response to "Command 0x1C: GetSoundCheckState" (page 279) and returns the current state of the sound check setting.

Table 4-93 `RetSoundCheckState` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		

Value	Bytes	Parameter
0x*NN*	1	`sndChkState`. The current state of the sound check setting. A value of 0x00 indicates that sound check is off; 0x01 indicates that it is on.
Packet footer specified in "Command Packets" (page 109).		

4.3.33 Command 0x1E: SetSoundCheckState

Lingo: 0x03 — Origin: Accessory

The accessory sets the state of the Apple device's sound check setting and optionally saves the previous sound check state to be restored on accessory detach. In response to this command, the Apple device sends an `iPodAck` packet with the status of the command.

Table 4-94 `SetSoundCheckState` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`sndChkState`. The new state of the sound check setting. A value of 0x00 turns off sound check off; 0x01 turns on sound check.
0x*NN*	1	`bRestoreOnExit`. Specifies whether the original sound check state is saved and restored on exit. See Table 4-75 (page 270)
Packet footer specified in "Command Packets" (page 109).		

4.3.34 Command 0x1F: GetTrackArtworkTimes

Lingo: 0x03 — Origin: Accessory

Note: This command has been superseded by the Extended Interface lingo "Command 0x004C: GetArtworkTimes" (page 458), which provides better functionality.

The accessory sends this command to the Apple device to request the list of artwork time offsets for a track. A 4-byte `trackIndex` specifies which track is to be selected. A 2-byte `formatID` indicates which type of artwork is desired. The format IDs that the Apple device supports can be obtained using the "Command 0x16: GetArtworkFormats" (page 275) command.

The 2-byte `artworkIndex` specifies where to begin searching for artwork. A value of 0 indicates that the Apple device should start with the first available artwork.

The 2-byte `artworkCount` specifies the maximum number of times to be returned. A value of -1 (0xFFFF) indicates that all artwork times from the specified index should be returned.

Table 4-95 `GetTrackArtworkTimes` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	`trackIndex`
0x*NN*	2	`formatID`
0x*NN*	2	`artworkIndex`
0x*NN*	2	`artworkCount`
Packet footer specified in "Command Packets" (page 109).		

4.3.35 Command 0x20: RetTrackArtworkTimes

Lingo: 0x03 — Origin: Apple device

Note: This command has been superseded by the Extended Interface lingo "Command 0x004D: RetArtworkTimes" (page 459), which provides better functionality.

The Apple device sends this command to the accessory in response to a "Command 0x1F: GetTrackArtworkTimes" (page 280) command. The accessory returns zero or more 4-byte records, one for each piece of artwork associated with the track and format specified by `GetTrackArtworkTimes`. Artwork times are expressed as offsets, in milliseconds, from the beginning of the track.

The number of records returned will be no greater than the number specified in the `GetTrackArtworkTimes` command. It may, however, be less than requested. This can happen if there are fewer pieces of artwork available than were requested, or if the Apple device is unable to place the full number in a single packet. Check the number of records returned against the results of `RetIndexedPlayingTrackInfo` with `infoType` 0x08 to ensure that all artwork has been received.

Table 4-96 `RetTrackArtworkTimes` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Time offset in milliseconds
0x*NN*	*NN*	The preceding time offset field may be repeated *NN* times.
Packet footer specified in "Command Packets" (page 109).		

4.3.36 Command 0x21: CreateGeniusPlaylist

Lingo: 0x03 — Origin: Accessory

The accessory sends this command to an Apple device to ask it to create a Genius playlist. The Apple device returns a Display Remote lingo `iPodAck` command, passing one of the responses listed in Table 4-98 (page 282). If the playlist is successfully created, it starts playing.

Note: This command may take up to 30 seconds to finish.

Table 4-97 `CreateGeniusPlaylist` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Track index: Playback index
Packet footer specified in "Command Packets" (page 109).		

Table 4-98 iPodAck responses to CreateGeniusPlaylist

iPodAck response	Meaning
0x00	Genius playlist is being created. When the playlist is complete, the Apple device sends an `iPodNotification` command for Command Complete; see Table 3-124 (page 189).
0x02	Genius playlist could not be created.
0x04	Track index not valid.
0x12	Genius information not available for that track (see "Command 0x22: IsGeniusAvailableForTrack" (page 282)).

4.3.37 Command 0x22: IsGeniusAvailableForTrack

Lingo: 0x03 — Origin: Accessory

The accessory sends this command to an Apple device to determine if Genius information is available for a given track. The Apple device returns a Display Remote lingo `iPodAck` command, passing one of the responses listed in Table 4-100 (page 283).

Table 4-99 `IsGeniusAvailableForTrack` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Track index: Playback index
Packet footer specified in "Command Packets" (page 109).		

Table 4-100 iPodAck responses to IsGeniusAvailableForTrack

iPodAck response	Meaning
0x00	Genius information is available. This does not guarantee that the Apple device can create a Genius playlist, because doing so depends on factors such as the availability of matching songs in the database.
0x04	Track index not valid.
0x12	Track index is valid, but Genius information is not available for that track.

4.4 Lingo 0x04: Extended Interface Lingo

Lingo 0x04 of the iAP is specified in detail in "Extended Interface Mode" (page 397).

4.5 Lingo 0x05: Accessory Power Lingo

Lingo 0x05 of the iAP is **deprecated**; do not use in new products. For historical information, see "Deprecated Lingo 0x05: Accessory Power Lingo" (page 547).

4.6 Lingo 0x06: USB Host Mode Lingo

All Apple devices can operate as devices controlled by an external USB host. Some Apple devices (see Table 1-6 (page 45)) can also act as hosts controlling external USB devices. This USB Host Mode is described in "USB Host Mode" (page 88). By default, the Apple device boots up in USB Device mode, ready to connect to a Macintosh or Windows computer for media data transfers. If the Apple device supports this lingo, the iAP commands specified in this section may be used to change it to USB Host mode.

Note: To work in USB Host Mode, the accessory must be using UART transport for iAP communication.

Table 4-101 (page 283) lists the iAP commands used to manage USB Host mode.

Table 4-101 USB Host mode commands

Cmd ID	Command name	Direction	Parameters:bytes
0x00	`AccessoryAck`	Acc to Dev	`{cmdStatus:1, cmdIDOrig:1}`
0x04	`NotifyUSBMode`	Dev to Acc	`{usbMode:1}`
0x80	`iPodAck`	Dev to Acc	`{cmdStatus:1, cmdIDOrig:1}`
0x81	`GetiPodUSBMode`	Acc to Dev	none

Cmd ID	Command name	Direction	Parameters:bytes
0x82	`RetiPodUSBMode`	Dev to Acc	`{usbMode:1}`
0x83	`SetiPodUSBMode`	Acc to Dev	`{usbMode:1}`

4.6.1 Command History of the USB Host Lingoes

Table 4-102 (page 284) shows the history of command changes in the USB Host Control and USB Host Mode lingoes:

Table 4-102 Command history of the USB Host Control and Host Mode lingoes

Lingo name	Version	Command changes	Features
USB Host Control	1.00	Add: 0x00-0x03	USB power state support
USB Host Mode	1.00	Deprecate: 0x00–0x03	See "Deprecated USB Host Control Commands" (page 549)
		Add: new 0x00, 0x04, and 0x80–0x83	Support USB Host mode on the Apple device

4.6.2 Command 0x00: AccessoryAck

Lingo: 0x06 — Origin: Accessory

The accessory sends this command in response to a command received from the Apple device. The accessory must send a response when a bad parameter is received, an unsupported or invalid command is received, or a command that does not return any data has finished.

Table 4-103 `AccessoryAck` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command status; see Table 3-6 (page 125).
0x*NN*	1	ID of the command being acknowledged
Packet footer specified in "Command Packets" (page 109).		

4.6.3 Command 0x04: NotifyUSBMode

Lingo: 0x06 — Origin: Apple device

The Apple device sends this command to the accessory to notify it that the Apple device's USB host/device role has changed. In response, the accessory must return an `AccessoryAck` command.

Table 4-104 `NotifyUSBMode` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`usbMode`: USB mode status; see Table 4-105 (page 285)
Packet footer specified in "Command Packets" (page 109).		

Table 4-105 USB Host mode status codes

Value	Description
0x00	Apple device USB interface is disabled or unavailable
0x01	Apple device USB interface is in Device Mode
0x02	Apple device USB interface is in Host Mode
0x03–0xFF	Reserved

4.6.4 Command 0x80: iPodAck

Lingo: 0x06 — Origin: Apple device

The Apple device sends this command in response to a command received from the accessory. This command is sent when a bad parameter is received, an unsupported or invalid command is received, or a command that does not return any data has finished.

Table 4-106 `iPodAck` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command status; see Table 3-6 (page 125).
0x*NN*	1	ID of the command being acknowledged
Packet footer specified in "Command Packets" (page 109).		

4.6.5 Command 0x81: GetiPodUSBMode

Lingo: 0x06 — Origin: Accessory

The accessory sends this command to determine the current Apple device USB Host mode status. In response, the Apple device returns a `RetiPodUSBMode` command.

Table 4-107 `GetiPodUSBMode` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.6.6 Command 0x82: RetiPodUSBMode

Lingo: 0x06 — Origin: Apple device

The Apple device sends this command in response to a `GetiPodUSBMode` command, passing the status of its USB Host mode.

Table 4-108 `RetiPodUSBMode` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`usbMode`: USB mode status; see Table 4-105 (page 285)
Packet footer specified in "Command Packets" (page 109).		

4.6.7 Command 0x83: SetiPodUSBMode

Lingo: 0x06 — Origin: Accessory

The accessory sends this command to set a new USB Host mode on the Apple device. In response, the Apple device returns an `iPodAck` command with the status of the operation. If the Apple device is already in the requested USB mode, the `iPodAck` command returns a success (0x00) status. If the requested USB mode cannot be set (for example, if the accessory requests USB Device mode while it uses hardware to force the Apple device into USB Host mode), the `iPodAck` command returns a Command Unavailable (0x10) status.

Table 4-109 `SetiPodUSBMode` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`usbMode`: USB mode to set; see Table 4-110 (page 287)
Packet footer specified in "Command Packets" (page 109).		

Table 4-110 SetiPodUSBMode codes

Value	Description
0x00-0x01	Reserved
0x02	Set Apple device USB interface to Host Mode
0x03–0xFF	Reserved

4.7 Lingo 0x07: RF Tuner Lingo

An external radio frequency tuner accessory can be attached to an Apple device to provide radio reception. When an Apple device successfully acknowledges an accessory's declaration of the RF Tuner lingo during the identification process, it makes a radio menu item available. Choosing this menu item displays the Apple device's tuner application. The application lets the user change the Apple device's music source and control the accessory's RF band and tuner frequency.

Note: If an Apple device contains an RF Tuner (such as the FM tuner in the 5G nano), the external tuner accessory will override it.

The RF Tuner lingo is used to pass control and state information between an Apple device and an RF tuner accessory. Normally the Apple device is the master and the accessory responds to commands from the Apple device. This means that the Apple device can initiate actions such as controlling tuner power, setting the tuner's band and frequency, initiating up or down frequency scans, and so on. An Apple device attached to an RF tuner accessory also stores station frequencies and other tuner state information.

Every RF tuner accessory must report all of its capabilities back to the attached Apple device, including support for at least one of the RF bands listed in Table 4-117 (page 293). Based on the capabilities reported by the RF tuner accessory, the Apple device's tuner application may choose to change its appearance to reflect the presence or absence of certain RF tuner features.

4.7.1 RF Tuner Accessory Design

RF tuner accessory devices for Apple devices must meet the following requirements:

- All RF tuner lingo commands require authentication.
- An Apple device can support only one RF tuner accessory at any time, and that accessory must be attached using the 30-pin connector.
- On reset or power up, the RF tuner accessory must be in low power mode, as defined in the section "Accessory Power Policy" in *MFi Accessory Hardware Specification*, with the tuner off and any audio output disabled.
- US devices must support the Europe/US FM band (87.5–108.0 MHz), with 200 KHz channel spacing and default 75 µsec deemphasis.
- EU/US devices must support the Europe/US FM band (87.5–108.0 MHz), with 100 KHz channel spacing and selectable 50 µsec or 75 µsec deemphasis.

- JP devices must support the Japan FM band (76.0–90.0 MHz), with 100 KHz channel spacing and selectable 50 µsec or 75 µsec deemphasis.

An RF tuner accessory may support US, EU, and/or JP requirements.

The following options apply to RF tuner devices:

- An RF tuner accessory may send asynchronous notification of state changes to an attached Apple device, but such notifications are not required.
- RF tuner devices are not expected to retain state information across interruptions of current from the Apple device resulting from the invocation of Hibernate mode.
- RF tuner devices may support HD radio.

4.7.2 RF Tuner Power

An attached RF tuner accessory will be notified of Apple device state changes such as transitions between Power On, Sleep, and Hibernate states. It is responsible for keeping its power consumption below the maximum allowed limits for each Apple device state. For details, see "Accessory Power Policy" in *MFi Accessory Hardware Specification*.

Current from the Apple device to the accessory is completely shut off during hibernation. When an Apple device wakes from hibernation and the accessory detects current from the Apple device on the Accessory Power line of the 30-pin connector, the accessory must wait at least 80 ms and then start the IDPS process. When an Apple device transitions from sleep to power on, it sends a `RequestIdentify` command to the accessory. In either case, the accessory must reidentify and reauthenticate itself, using IDPS (or `IdentifyDeviceLingoes` if the Apple device does not support IDPS).

RF Tuner accessories can enter Intermittent High Power mode, as specified in the "Accessory Power Policy" section of *MFi Accessory Hardware Specification*, after receiving a `SetTunerCtrl` command specifying power-on. They must reduce their power consumption to Low Power mode within 1 second of receiving a `SetTunerCtrl` command specifying power-off. Regardless of the power state specified by the Apple device, all devices must comply with the Apple device state changes cited in the previous paragraph.

4.7.3 RF Tuner Lingo Commands

Table 4-111 (page 288) summarizes the RF Tuner commands (lingo 0x07).

IMPORTANT: The accessory must not exceed the command timings listed in Table 4-2 (page 212) when responding to RF Tuner commands from the Apple device.

Table 4-111 RF Tuner lingo command summary

Command	ID	Direction	Data length	Protocol version	Authentication required
`AccessoryAck`	0x00	Acc to Dev	4 or 8	1.00	Yes
`GetTunerCaps`	0x01	Dev to Acc	2	1.00	Yes

Command	ID	Direction	Data length	Protocol version	Authentication required
RetTunerCaps	0x02	Acc to Dev	8	1.00	Yes
GetTunerCtrl	0x03	Dev to Acc	2	1.00	Yes
RetTunerCtrl	0x04	Acc to Dev	3	1.00	Yes
SetTunerCtrl	0x05	Dev to Acc	3	1.00	Yes
GetTunerBand	0x06	Dev to Acc	2	1.00	Yes
RetTunerBand	0x07	Acc to Dev	3	1.00	Yes
SetTunerBand	0x08	Dev to Acc	3	1.00	Yes
GetTunerFreq	0x09	Dev to Acc	2	1.00	Yes
RetTunerFreq	0x0A	Acc to Dev	7	1.00	Yes
SetTunerFreq	0x0B	Dev to Acc	6	1.00	Yes
GetTunerMode	0x0C	Dev to Acc	2	1.00	Yes
RetTunerMode	0x0D	Acc to Dev	3	1.00	Yes
SetTunerMode	0x0E	Dev to Acc	3	1.00	Yes
GetTunerSeekRssi	0x0F	Dev to Acc	2	1.00	Yes
RetTunerSeekRssi	0x10	Acc to Dev	3	1.00	Yes
SetTunerSeekRssi	0x11	Dev to Acc	3	1.00	Yes
TunerSeekStart	0x12	Dev to Acc	3	1.00	Yes
TunerSeekDone	0x13	Acc to Dev	7	1.00	Yes
GetTunerStatus	0x14	Dev to Acc	2	1.00	Yes
RetTunerStatus	0x15	Acc to Dev	3	1.00	Yes
GetStatusNotifyMask	0x16	Dev to Acc	2	1.00	Yes
RetStatusNotifyMask	0x17	Acc to Dev	3	1.00	Yes
SetStatusNotifyMask	0x18	Dev to Acc	3	1.00	Yes
StatusChangeNotify	0x19	Acc to Dev	3	1.00	Yes
GetRdsReadyStatus	0x1A	Dev to Acc	2	1.00	Yes
RetRdsReadyStatus	0x1B	Acc to Dev	6	1.00	Yes
GetRdsData	0x1C	Dev to Acc	3	1.00	Yes

Command	ID	Direction	Data length	Protocol version	Authentication required
RetRdsData	0x1D	Acc to Dev	0x*NN*	1.00	Yes
GetRdsNotifyMask	0x1E	Dev to Acc	2	1.00	Yes
RetRdsNotifyMask	0x1F	Acc to Dev	6	1.00	Yes
SetRdsNotifyMask	0x20	Dev to Acc	6	1.00	Yes
RdsReadyNotify	0x21	Acc to Dev	0x*NN*	1.00	Yes
Reserved	0x22–0x24	N/A	N/A	N/A	N/A
GetHDProgramServiceCount	0x25	Dev to Acc	0	1.01	Yes
RetHDProgramServiceCount	0x26	Acc to Dev	1	1.01	Yes
GetHDProgramService	0x27	Dev to Acc	0	1.01	Yes
RetHDProgramService	0x28	Acc to Dev	1	1.01	Yes
SetHDProgramService	0x29	Dev to Acc	1	1.01	Yes
GetHDDataReadyStatus	0x2A	Dev to Acc	0	1.01	Yes
RetHDDataReadyStatus	0x2B	Acc to Dev	4	1.01	Yes
GetHDData	0x2C	Dev to Acc	1	1.01	Yes
RetHDData	0x2D	Acc to Dev	0x*NN*	1.01	Yes
GetHDDataNotifyMask	0x2E	Dev to Acc	0	1.01	Yes
RetHDDataNotifyMask	0x2F	Acc to Dev	4	1.01	Yes
SetHDDataNotifyMask	0x30	Dev to Acc	4	1.01	Yes
HDDataReadyNotify	0x31	Acc to Dev	0x*NN*	1.01	Yes
Reserved	0x32–0xFF	N/A	N/A	N/A	N/A

4.7.4 Command History of the RF Tuner Lingo

Table 4-112 (page 290) shows the history of changes to the RF Tuner lingo.

Table 4-112 RF Tuner lingo command history

Lingo version	Command changes	Features
1.00	Add: 0x00–0x21	RF tuner control and support

Lingo version	Command changes	Features
1.01	Update: 0x02, 0x04, 0x05, 0x07, 0x0D, 0x0E, 0x12, 0x13, 0x15, 0x17, 0x18, 0x19, 0x1B, 0x1C, 0x1D, 0x1F, 0x20, and 0x21 Add: 0x25–0x31	Support for HD radio, AM tuning, FM wide band

4.7.5 Command 0x00: AccessoryAck

Lingo: 0x07 — Origin: Accessory

This command must be sent by an RF tuner accessory on completion of a command received from an Apple device. An `AccessoryAck` response must also be sent if a bad parameter is received, an unsupported or invalid command is received, or a command that does not return any data has finished.

Note: Certain fields of the RF tuner lingo `AccessoryAck` command packet are included only if the command status value is 0x06. See Table 4-113 (page 291) and Table 4-114 (page 291).

Table 4-113 RF tuner lingo `AccessoryAck` packet with command status not 0x06

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command status; see Table 3-6 (page 125).
0x*NN*	1	ID of the command for which the response is being sent
Packet footer specified in "Command Packets" (page 109).		

Table 4-114 RF tuner lingo `AccessoryAck` packet with command status equal to 0x06

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x06	1	Command status: Command pending
0x*NN*	1	`cmdIDOrig`: ID of the command for which the response is being sent
0x*NN*	4	`cmdPendTime`. Total timeout in milliseconds for the pending command to complete.
Packet footer specified in "Command Packets" (page 109).		

Note: A value of 0x06 must be returned in the command status byte of the RF tuner `AccessoryAck` packet for commands that require more than 100 ms to complete. In this case, a `cmdPendTime` parameter is included in the `AccessoryAck` packet. If the completion status is not returned by the total number of milliseconds specified by these bytes, the Apple device will assume that a command failure has occurred and will retry the command or otherwise recover from the error. The RF tuner accessory should not return any response to the command after this timeout period has expired.

4.7.6 Command 0x01: GetTunerCaps

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to query an attached RF tuner accessory's capabilities and determine what features the accessory supports. In response, the accessory must send a `RetTunerCaps` command with a payload specifying its capabilities.

Table 4-115 `GetTunerCaps` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.7.7 Command 0x02: RetTunerCaps

Lingo: 0x07 — Origin: Accessory

This command is sent by an RF tuner accessory in response to a `GetTunerCaps` command sent by an Apple device. The command transmits a payload indicating which RF tuner capabilities it supports.

Table 4-116 `RetTunerCaps` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Tuner capabilities (See Table 4-117 (page 293))
0x01	1	Reserved; set to 0x01
0x00	1	Reserved; set to 0x00
Packet footer specified in "Command Packets" (page 109).		

Note: The capabilities bits returned in the `RetTunerCaps` payload (bytes 5-8) correspond to the band, control, mode, and status commands of the RF tuner lingo. The Apple device will enable features or use commands only if the RF tuner accessory sets the associated capabilities bits when sending this command.

Table 4-117 RF tuner accessory capabilities payload

Bits	Capability
00	AM band, Worldwide (520-1710 KHz) capable
01	FM band, Europe/US (87.5–108.0 MHz) capable
02	FM band, Japan (76.0–90.0 MHz) capable
03	FM band, Wide (76.0-108.0 MHz) capable
04	HD Radio capable; may be set only if bit 01 is set
07:05	Reserved; set to 0
08	Tuner power on/off control capable
09	Status change notification capable
15:10	Reserved; set to 0
17:16	Minimum FM resolution ID bits (see Table 4-118 (page 294))
18	Tuner seek up/down capable
19	Tuner seek RSSI threshold capable (must not be set if bit 18 is 0)
20	Force monophonic mode capable
21	Stereo blend capable
22	FM Tuner deemphasis select capable
23	AM tuner resolution 9KHz (0=10KHz only) capable
24	Radio Data System (RDS/RBDS) data capable
25	Tuner channel RSSI indication capable
26	Stereo source indicator capable
27	RDS/RDBS Raw mode capable
31:28	Reserved; set to 0

Note: Bit 04 (HD Radio capable), listed in Table 4-117 (page 293), may be set in addition to the FM band bits (00-03), but only if bit 01 is set. This bit indicates that the radio is capable of receiving HD signals on an existing Europe/US AM or FM band.

Table 4-118 Minimum FM resolution ID bits

ID bits	Description
00	200 kHz capable
01	100 kHz capable
10	50 kHz capable
11	Reserved

Note: The minimum FM resolution bits (bits 17-16) should report the smallest FM tuner resolution that the RF tuner accessory supports.

4.7.8 Command 0x03: GetTunerCtrl

Lingo: 0x07 — Origin: Apple device

An Apple device sends this command to an RF tuner accessory to get the accessory's control state. In response, the accessory must send a `RetTunerCtrl` command with its current control state.

Table 4-119 `GetTunerCtrl` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.7.9 Command 0x04: RetTunerCtrl

Lingo: 0x07 — Origin: Accessory

An RF tuner accessory uses this command to send its current accessory control state in response to a `GetTunerCtrl` command from an Apple device. Tuner control bits may be set only if the corresponding capabilities bits have been set in a previous `RetTunerCaps` command.

Table 4-120 `RetTunerCtrl` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Tuner control state (see Table 4-121 (page 295))
Packet footer specified in "Command Packets" (page 109).		

Table 4-121 Tuner control state bits

Bit	Description
0	RF tuner accessory power draw is on (1) or off (0). When RF tuner accessory power draw is off, the accessory should rest in the lowest power state that still allows iAP commands to be received and processed.
1	Status change notification is enabled (1) or disabled (0). When status change notification is disabled, the Apple device assumes that the accessory will send no asynchronous `StatusChangeNotify` commands. The Apple device may poll the accessory for changes.
2	Reserved
3	RDS/RBDS Raw mode enabled
7:4	Reserved

4.7.10 Command 0x05: SetTunerCtrl

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to control an RF tuner accessory's state. In response, the accessory must return an `AccessoryAck` command with the command status. RF tuner state information, such as tuner frequency, band, and so on, must be preserved by the accessory across tuner on and off cycles, assuming current is available from the Apple device.

When a tuner accessory enters Intermittent High Power mode, as defined in the section "Accessory Power Policy" in *MFi Accessory Hardware Specification*, the Apple device enables its accessory high power output before sending a `SetTunerCtrl` command to turn on tuner power. When tuner power is switched on, the accessory's power consumption may rise to the maximum declared during its identification process.

The Apple device accessory high power remains on until a subsequent `SetTunerCtrl` command to turn off tuner power has finished. The accessory must reduce its power consumption to low power mode within 1 second after receiving this command. When the tuner power is turned off, it must disable its audio output and reduce its power consumption to Low Power mode.

Note: The Apple device `NotifyiPodStateChange` command overrides this command for Apple device transitions to the Sleep or Hibernate state. Even if this command has let the RF tuner draw current from the Apple device, the accessory must reduce its power consumption to low power mode on receiving an Apple device state change notification that specifies a transition to Sleep or Hibernate. The only Apple device power state in which an RF tuner accessory requesting high power can consume more than low power is the Power On state.

Table 4-122 `SetTunerCtrl` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Tuner control bits (see Table 4-123 (page 296))
Packet footer specified in "Command Packets" (page 109).		

Table 4-123 Tuner control bits

Bit	Description
0	Turn RF tuner current draw from the Apple device on (1) or off (0). When RF tuner current draw is turned off, the accessory should rest in the lowest power state that still allows iAP commands to be received and processed.
1	Enable (1) or disable (0) status change notification. When status change notification is disabled, the Apple device assumes that the accessory will send no asynchronous `StatusChangeNotify` commands. The Apple device may poll the accessory for changes.
2	Reserved
3	Enable RDS/RBDS Raw mode. When enabled, raw RDS/RBDS data format is used by the RDS data commands (commands 0x1A-0x21). When disabled, parsed RDS data is used by the RDS data commands.
7:4	Reserved

4.7.11 Command 0x06: GetTunerBand

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to get an RF tuner accessory's RF band information. The accessory must respond by returning a `RetTunerBand` command.

Table 4-124 `GetTunerBand` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		

Value	Bytes	Parameter
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.7.12 Command 0x07: RetTunerBand

Lingo: 0x07 — Origin: Accessory

This command is sent by an RF tuner accessory to report its tuner band state in response to an Apple device's `GetTunerBand` command. If the RF tuner current draw from the Apple device is off, it should return its last active band state.

Note: Tuner band state information must be consistent with the accessory's capabilities, as previously reported by a `RetTunerCaps` command.

Table 4-125 `RetTunerBand` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Tuner band state information (see Table 4-126 (page 297)):
Packet footer specified in "Command Packets" (page 109).		

Table 4-126 Tuner band state IDs

ID	Description
0x00	AM band worldwide (520-1710 KHz)
0x01	Europe/US FM band (87.5–108.0 MHz)
0x02	Japan FM band (76.0–90.0 MHz)
0x03	FM wide band (76.0-108.0 MHz)
0x04–0xFF	Reserved

4.7.13 Command 0x08: SetTunerBand

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to an RF tuner accessory to set its tuner band. In response, the accessory must send the Apple device an `AccessoryAck` command with the command status. The command status must be reported as command failed (command status 0x02) if RF tuner current draw from the Apple device is not on.

Table 4-127 `SetTunerBand` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Tuner band to be set (see Table 4-126 (page 297))
Packet footer specified in "Command Packets" (page 109).		

Note: The tuner band setting requested by the Apple device will be consistent with the RF tuner accessory's capabilities, as previously reported by a `RetTunerCaps` command.

4.7.14 Command 0x09: GetTunerFreq

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to get an RF tuner accessory's current accessory tuner frequency and signal strength level. In response, the accessory must send the Apple device a `RetTunerFreq` command.

Table 4-128 `GetTunerFreq` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.7.15 Command 0x0A: RetTunerFreq

Lingo: 0x07 — Origin: Accessory

This command is sent by an RF tuner accessory in response to a `GetTunerFreq` or `SetTunerFreq` command received from an Apple device. The tuner frequency is expressed in kilohertz: for example, 76000 for 76.0 MHz, 87500 for 87.5 MHz, or 107900 for 107.9 MHz. Valid ranges are 87500 to 107900 for the European/US FM band and 76000 to 89900 for the Japanese FM band.

If the RF tuner accessory's capabilities includes tuner channel RSSI indication, field `rssiLevel` of the command parameters may be a number greater than zero; otherwise it must be set to 0x00. The tuner RSSI level must be normalized to a value between 0 (minimum) and 255 (maximum). If RF tuner current draw from the Apple device is off, the last active tuner frequency and signal strength (if applicable) should be sent.

IMPORTANT: The accessory must send this command as quickly as possible in response to a `GetTunerFreq` command from the Apple device. Tuning will fail if the response time is longer than 200 ms.

Table 4-129 `RetTunerFreq` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Tuner frequency in kilohertz
0x*NN*	1	`rssiLevel`: Tuner channel received signal strength; may range from 0x00 (no signal present or RSSI indication not supported) to 0xFF (maximum RSSI signal strength).
Packet footer specified in "Command Packets" (page 109).		

Note: The tuner channel received signal strength value is valid only if the tuner RSSI bit was set in a previous `RetTunerCaps` command.

4.7.16 Command 0x0B: SetTunerFreq

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to an RF tuner accessory to select a specific tuner frequency within the current tuner band. The tuner frequency is expressed in kilohertz; valid ranges are 87500 to 107900 for the European/US FM band and 76000 to 89900 for the Japanese FM band. In response, the accessory must send a `RetTunerFreq` command with the new tuner frequency and RSSI level (if the accessory supports RSSI). The requested frequency should be within the currently selected tuner band. The command status must be reported as command failed (command status 0x02) if RF tuner current draw from the Apple device is not on.

Table 4-130 `SetTunerFreq` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Tuner frequency in kilohertz
Packet footer specified in "Command Packets" (page 109).		

4.7.17 Command 0x0C: GetTunerMode

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to an RF tuner accessory to get the current tuner mode state from the accessory. In response, the accessory must send the Apple device a `RetTunerMode` command.

Table 4-131 `GetTunerMode` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.7.18 Command 0x0D: RetTunerMode

Lingo: 0x07 — Origin: Accessory

This command is sent by an accessory in response to a `GetTunerMode` command received from an Apple device. The tuner mode bits returned are valid only if the associated mode bits returned by a previous `RetTunerCaps` command were set. If tuner power is off, the last active tuner mode information should be returned.

Table 4-132 `RetTunerMode` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Tuner mode status (see Table 4-133 (page 300))
Packet footer specified in "Command Packets" (page 109).		

Table 4-133 RF Tuner mode status bits

Bits	Description
1:0	FM tuner resolution (see Table 4-118 (page 294))
2	Tuner is currently seeking up or down
3	Tuner is currently seeking with an RSSI minimum threshold enabled
4	Monophonic mode is forced (1) or stereo is allowed (0)
5	Stereo blend is enabled (valid only if bit 4 is 0)
6	FM Tuner deemphasis is 50 µsec (1) or 75 µsec (0)
7	AM tuner resolution (1 = 9 KHz, 0 = 10 KHz)

4.7.19 Command 0x0E: SetTunerMode

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to an RF tuner accessory to set the current tuner mode. In response, the accessory must send an RF tuner lingo `AccessoryAck` command with the command status. The command status must be reported as command failed (command status 0x02) if RF tuner current draw from the Apple device is not on.

Table 4-134 `SetTunerMode` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Tuner mode to be set (see Table 4-135 (page 301))
Packet footer specified in "Command Packets" (page 109).		

Table 4-135 Set RF Tuner mode bits

Bits	Description
1:0	Set FM tuner resolution (see Table 4-118 (page 294))
2-3	Reserved; set to 0
4	Force monophonic mode (1) or allow stereo (0)
5	Enable stereo blend (valid only if bit 4 is 0); this blends the source left and right channels to improve the perceived signal quality while still retaining some stereo image separation
6	Set FM Tuner deemphasis to 50 µsec (1) or 75 µsec (0)
7	Set AM tuner resolution (1 = 9 KHz, 0 = 10 KHz)

Note: `SetTunerMode` must not be used to start tuner seeking or set the seek type. The `TunerSeekStart` command must be used instead.

4.7.20 Command 0x0F: GetTunerSeekRssi

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to get the current tuner seek threshold value from an RF tuner accessory. In response, the accessory must send a `RetTunerSeekRssi` command with the seek RSSI threshold, if supported by the accessory. If seek RSSI is not supported, the RF tuner accessory must return an `AccessoryAck` command with failure status.

Table 4-136 `GetTunerSeekRssi` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		

Value	Bytes	Parameter
Packet footer specified in "Command Packets" (page 109).		

4.7.21 Command 0x10: RetTunerSeekRssi

Lingo: 0x07 — Origin: Accessory

This command is sent by an accessory in response to a `GetTunerSeekRssi` command received from an Apple device, assuming the accessory supports the seek RSSI capability. Its threshold value represents the minimum signal strength that allows a tuner channel to be recognized during the seek process; it is normalized to a value between 0 (minimum) and 255 (maximum). If current draw from the Apple device is off, the last active RSSI threshold value should be sent.

Table 4-137 `RetTunerSeekRssi` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	RSSI threshold for seeking action
Packet footer specified in "Command Packets" (page 109).		

4.7.22 Command 0x11: SetTunerSeekRssi

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to an RF tuner accessory that supports the seek RSSI capability, to set the accessory's seek RSSI signal strength threshold. The threshold value is the minimum signal strength that allows a tuner channel to be recognized during the seek process; it is normalized to a range between 0 (minimum) and 255 (maximum).

Table 4-138 `SetTunerSeekRssi` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	RSSI threshold for seeking action
Packet footer specified in "Command Packets" (page 109).		

4.7.23 Command 0x12: TunerSeekStart

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to an RF tuner accessory that supports the tuner seek up/down capability (as reported by a previous `RetTunerCaps` command), to initiate seeking of a specified type. The returned `AccessoryAck` command status must be reported as command failed (command status 0x02) if RF tuner current draw from the Apple device is not on. Seeking operations that use an RSSI threshold are initiated only if the RF tuner accessory's seek RSSI threshold capability is also supported, as reported by the `RetTunerCaps` command.

Table 4-139 `TunerSeekStart` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	ID of tuner seeking operation (see Table 4-140 (page 303) and Note below)
Packet footer specified in "Command Packets" (page 109).		

Note: An accessory that declares support for HD radio by setting its capability bit 04 (see Table 4-117 (page 293)) must also support HD seeks. If the accessory receives a seek code it does not support, it must return an `AccessoryAck` command with Bad Parameter status (0x04).

Table 4-140 Tuner seeking operations

ID	Operation	Use RSSI threshold
0x00	No seek operation (cancel seek operation, if active)	N/A
0x01	Seek up from beginning of band	No
0x02	Seek down from end of band	No
0x03	Seek up from current frequency	No
0x04	Seek down from current frequency	No
0x05	Seek up from beginning of band	Yes
0x06	Seek down from end of band	Yes
0x07	Seek up from current frequency	Yes
0x08	Seek down from current frequency	Yes
0x09	Seek up from beginning of band for an HD signal	No
0x0A	Seek down from end of band for an HD signal	No
0x0B	Seek up from current frequency for an HD signal	No
0x0C	Seek down from current frequency for an HD signal	No
0x0D	Seek up from beginning of band for an HD signal	Yes

ID	Operation	Use RSSI threshold
0x0E	Seek down from end of band for an HD signal	Yes
0x0F	Seek up from current frequency for an HD signal	Yes
0x10	Seek down from current frequency for an HD signal	Yes
0x11–0xFF	Reserved	N/A

An analog seek operation (operation 0x01-0x08) will seek any frequency that contains a valid signal including HD signals. An HD seek operation (operation 0x09-0x10) skips channels that are not HD and stops only on channels with HD signals present. The HD seek completes when either of two conditions is satisfied:

- An HD channel that satisfies the criteria of the tuner's seek function was located within the band. This may result in moving one or more channel spacings and wrapping around one end of the band.
- No HD channel that satisfies the criteria of the tuner's seek function was located within the band, and the seek has traversed the entire band and wrapped back to the original tuner frequency.

A seek operation using an RSSI threshold completes when either of two conditions is satisfied:

- A channel was located within the band that satisfies the minimum RSSI threshold level.
- No channel was located within the band that satisfies the minimum the RSSI threshold level. The seek has traversed the entire band and wrapped back to the beginning tuner frequency without locating a valid channel. If no channel is found, it may indicate that the threshold is too high for the current radio reception area.

A seek operation using no RSSI threshold completes when either of two conditions is satisfied:

- A channel was located within the band that satisfies the criteria of the tuner's seek function. This may result in moving one or more channel spacings and wrapping around at the band ends.
- No channel was located within the band that satisfies the criteria of the tuner's seek function and the seek has traversed the entire band and wrapped back to the beginning tuner frequency without locating a valid channel.

An `AccessoryAck` command with command pending status must be returned for seek operations requiring more than 100 ms to complete. It indicates the maximum time required for the accessory to complete the requested scan type. When the requested seek operation is completed (either successfully or unsuccessfully), the accessory must respond with a `TunerSeekDone` command indicating the seek operation status. If the accessory does not support tuner seek (with RSSI) operations or if tuner power is off, an `AccessoryAck` command with error status must be returned.

When a cancel seek operation is requested and a seek operation is active, the seek operation stops at the current seek channel and the accessory responds with a `TunerSeekDone` command indicating the frequency and RSSI of the channel.

4.7.24 Command 0x13: TunerSeekDone

Lingo: 0x07 — Origin: Accessory

In response to a `TunerSeekStart` command from an Apple device, an RF tuner accessory must send this command to the Apple device after the seek operation has finished. It reports the current tuner frequency, which is assumed to be the result of the seek operation. If the accessory supports a tuner channel RSSI indication capability (as reported by a previous `RetTunerCaps` command), the `rssiLevel` value shows the current channel's RSSI signal strength level. If no channel was found, a tuner frequency value of 0xFFFFFFFF must be reported. The received signal strength value must be normalized to a range from 0x00 (no signal or the accessory does not support RSSI indication) to 0xFF (maximum signal strength).

If an HD signal seek was requested, the tuner should tune to the next frequency containing HD content but it should not select an analog or HD program service. The Apple device will perform the program selection operation separately. See "Command 0x28: RetHDProgramService" (page 317) and "Command 0x29: SetHDProgramService" (page 318).

Table 4-141 `TunerSeekDone` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Tuner frequency in kilohertz
0x*NN*	1	Tuner channel RSSI received signal strength value
Packet footer specified in "Command Packets" (page 109).		

4.7.25 Command 0x14: GetTunerStatus

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to an RF tuner accessory to get its current tuner status state; in response, the accessory must send a `RetTunerStatus` command. The command can be used to poll the accessory's status if the accessory does not support status change notifications. After the accessory's status is reported its status bits must be cleared, so that an immediate reread will return no active status.

Table 4-142 `GetTunerStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.7.26 Command 0x15: RetTunerStatus

Lingo: 0x07 — Origin: Accessory

This command is sent by an RF tuner accessory in response to a `GetTunerStatus` command received from an Apple device. The tuner status bits returned are valid only if the status capability bits returned by a previous `RetTunerCaps` command were set. If the tuner power is off, the last active tuner status information should be returned.

Table 4-143 `RetTunerStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Tuner status bits (see Table 4-144 (page 306))
Packet footer specified in "Command Packets" (page 109).		

Note: An accessory that declares support for HD radio by setting its capability bit 04 must return valid RF tuner status bits; see Table 4-117 (page 293).

Table 4-144 RF tuner status bits

Bit	Description
0	RDS/RBDS data is received, ready to read
1	Tuner channel RSSI level has changed
2	Stereo source indicator state; 1 for a stereo signal source, 0 for monophonic
3	HD signal present
4	HD digital audio present
5	HD data is received; ready to read
7:6	Reserved

4.7.27 Command 0x16: GetStatusNotifyMask

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to an RF tuner accessory to get the status notification mask from the accessory. This mask indicates which state changes will invoke a notification change command from the accessory. In response, the accessory must send the Apple device a `RetStatusNotifyMask` command.

Table 4-145 `GetStatusNotifyMask` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		

Value	Bytes	Parameter
Packet footer specified in "Command Packets" (page 109).		

4.7.28 Command 0x17: RetStatusNotifyMask

Lingo: 0x07 — Origin: Accessory

This command must be returned by the accessory in response to the `GetStatusNotifyMask` command. The status notification mask indicates which state changes will invoke a notification change command from the accessory. However, its bit values are valid only if the corresponding capabilities bits returned by a previous `RetTunerCaps` command were set. A mask bit value of 1 indicates that notification is enabled; a value of 0 indicates notification is disabled or not supported.

Table 4-146 `RetStatusNotifyMask` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Status notification mask (see Table 4-147 (page 307))
Packet footer specified in "Command Packets" (page 109).		

Note: An accessory that declares support for HD radio by setting its capability bit 04 (see Table 4-117 (page 293)) must support status notifications.

Table 4-147 Status notification mask bits

Bit	Description
0	RDS/RBDS data-ready change notify enabled (ignored if `SetRdsNotifyMask` sets `rdsMask` to a value other than zero)
1	Tuner channel RSSI-level change notify enabled
2	Stereo indicator state change notify enabled
3	HD signal present notification enabled
4	HD digital audio present notification enabled
5	HD data ready notification enabled
7:6	Reserved

4.7.29 Command 0x18: SetStatusNotifyMask

Lingo: 0x07 — Origin: Apple device

The Apple device sends this command to an RF tuner accessory to set the status change notification mask. The status notification mask indicates which state changes will invoke a notification change command from the accessory. However, its bit values are valid only if the corresponding capabilities bits returned by a previous `RetTunerCaps` command were set. A mask bit value of 1 indicates that notification is enabled; a value of 0 indicates notification is disabled or not supported.

Note: For RDS/RBDS data-ready changes, the `SetRdsNotifyMask` command can override the notification status mask bit. If the `SetRdsNotifyMask` notification mask is set with a nonzero mask, then any enabled RDS/RBDS data notifications will be sent using the `RdsReadyNotify` command instead of the `StatusChangeNotify` command. For HD data ready changes, the `SetHDDataNotifyMask` command can override the HD data notification status mask bit. If the `SetHDDataNotifyMask` notification mask is set with a nonzero mask, then any enabled HD data notifications must be sent using the `HDDataReadyNotify` command instead of the `StatusChangeNotify` command.

Table 4-148 `SetStatusNotifyMask` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Status notification mask (see Table 4-149 (page 308))
Packet footer specified in "Command Packets" (page 109).		

Table 4-149 Status notification mask setting bits

Bit	Description
0	Enable RDS/RBDS data-ready change notification (ignored if `SetRdsNotifyMask` sets `rdsMask` to a value other than zero)
1	Enable tuner channel RSSI-level change notification
2	Enable stereo-indicator state change notification
3	Enable HD signal present notification
4	Enable HD digital audio present notification
5	Enable HD data ready notification
7:6	Reserved

4.7.30 Command 0x19: StatusChangeNotify

Lingo: 0x07 — Origin: Accessory

This command must be sent asynchronously by an RF tuner accessory to an attached Apple device to report each enabled status change. The Apple device enables specific RF tuner status change notifications using the `SetStatusNotifyMask` command. After a notification has been sent, the status bits should be automatically cleared so they are ready to receive the next status change.

Note: Tuner channel RSSI-level change and stereo-indicator state change notifications must not occur more than once every 100 ms.

Table 4-150 `StatusChangeNotify` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Status change ID bits (see Table 4-151 (page 309))
Packet footer specified in "Command Packets" (page 109).		

Note: For RDS/RBDS data-ready changes, the `SetRdsNotifyMask` command can override the notification status mask bit. If the `SetRdsNotifyMask` notification mask is set with a nonzero mask, then any enabled RDS/RBDS data notifications will be sent using the `RdsReadyNotify` command instead of the `StatusChangeNotify` command. Similarly, for HD data-ready changes, the `SetHDDataNotifyMask` command can override the HD data notification status mask bit. If the `SetHDDataNotifyMask` notification mask is set with a nonzero mask, then any enabled HD data notifications will be sent using the `HDDataReadyNotify` command instead of the `StatusChangeNotify` command.

Table 4-151 Status change ID bits

Bit	Description
0	RDS/RBDS data-ready change (valid only if bit 0 of `statusMask` is 1 and `SetRdsNotifyMask` sets `rdsMask` to 0x0)
1	Tuner channel RSSI-level change
2	Stereo indicator state change
3	HD signal present
4	HD digital audio present
5	HD data is received; ready to read
7:6	Reserved

4.7.31 Command 0x1A: GetRdsReadyStatus

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to an RF tuner accessory to get the accessory's current RDS/RBDS data-ready status. It can be used to poll the accessory's RDS/RBDS data-ready status without having to enable RDS/RBDS data-ready notifications. This command is usable only if the `RetTunerCaps` capability bits report has indicated that the accessory supports RDS/RBDS data reception and parsing. In response, the accessory must send a `RetRdsReadyStatus` command.

Table 4-152 `GetRdsReadyStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.7.32 Command 0x1B: RetRdsReadyStatus

Lingo: 0x07 — Origin: Accessory

This command must be sent by an RF tuner accessory in response to a `GetRdsReadyStatus` command received from an Apple device. Its status value indicates which RDS/RBDS data values are available to be read. All status bits must remain set on the accessory until the Apple device sends a `GetRdsData` command to read the data. If the accessory does not support RDS/RBDS data, it must send an `rdsReady` data-ready status of 0 to indicate that no data is ready.

Table 4-153 `RetRdsReadyStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	RDS/RBDS data-ready status (see Table 4-154 (page 310) and Table 4-155 (page 310))
Packet footer specified in "Command Packets" (page 109).		

Table 4-154 RDS/RBDS data-ready status bits, parsed mode

Bit	Description
03-00	Reserved; set to zeros
04	RadioText (RT) data-ready
29-05	Reserved; set to zeros
30	Program Service Name (PSN) data-ready
31	Reserved; set to zero

Table 4-155 RDS/RBDS data-ready status bits, raw mode

Bit	Description
04-00	Reserved

Bit	Description
05	RDS/RBDS group data ready
31-06	Reserved

4.7.33 Command 0x1C: GetRdsData

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to get raw or unparsed RDS/RBDS data from an RF tuner accessory that supports the RDS/RBDS data capability. If the accessory supports RDS/RBDS and has data-ready, it must send a `RetRdsData` command. If the accessory does not support RDS/RBDS or does not have the specified data type ready, it must return an `AccessoryAck` command with command failure status.

Table 4-156 `GetRdsData` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	RDS/RBDS data type (`rdsDataType`) ID (see Table 4-157 (page 311) and Table 4-158 (page 311))
Packet footer specified in "Command Packets" (page 109).		

Table 4-157 RDS/RBDS data type IDs, parsed mode

ID Bytes	Description
0x00–0x03	Reserved
0x04	RadioText (RT)
0x05–0x1D	Reserved
0x1E	Program Service Name (PSN)
0x1F–0xFF	Reserved

Table 4-158 RDS/RBDS data type IDs, raw mode

ID Bytes	Description
0x00–0x04	Reserved
0x05	RDS/RBDS group data ready
0x06–0xFF	Reserved

4.7.34 Command 0x1D: RetRdsData

Lingo: 0x07 — Origin: Accessory

This command is sent by an RF tuner accessory that has an RDS/RBDS data group ready in response to a `GetRdsData` command received from an Apple device. If the accessory does not support the RDS/RBDS capability or does not have the specified data type ready, it must return an `AccessoryAck` command with command failure status (command status 0x02).

Table 4-159 `RetRdsData` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`rdsDataType` (see Table 4-160 (page 312))
0x*NN*	*NN*	`rdsData` (see Table 4-160 (page 312))
Packet footer specified in "Command Packets" (page 109).		

Table 4-160 `rdsDataType` bytes and `rdsData` formats

rdsDataType bytes	Description	rdsData format
0x00–0x03	Reserved	N/A
0x04	RadioText (RT)	`charSet`:1 (see Table 4-161 (page 312)), `radioText`:64
0x05–0x1D	Reserved	N/A
0x1E	Program Service Name (PSN)	`charSet`:1 (see Table 4-161 (page 312)), `progSvcName`:8
0x1F–0xFF	Reserved	N/A

Note: The maximum length of a Radio Text is 64 bytes and that of a Program Service Name is 8 bytes.

Table 4-161 `rdsData` character set IDs

ID	Character set
0x00	Latin-based languages
0x01	Cyrillic- and Greek-based languages
0x02	Arabic- and Hebrew-based languages
0x03–0xFF	Reserved

When RDS/RBDS Raw mode is enabled, the data shown in Table 4-162 (page 313) is returned for the rdsDataType 0x05 (RDS/RBDS group data ready).

Table 4-162 RDS/RBDS group data-ready data

Byte	Description
0-1	16 bit value for Block A of RDS/RBDS group data
2-3	16 bit value for Block B of RDS/RBDS group data
4-5	16 bit value for Block C of RDS/RBDS group data
6-7	16 bit value for Block D of RDS/RBDS group data
8	Block errors byte (see Table 4-163 (page 313))

Table 4-163 Block errors byte encoding

Byte	Description	
1:0	Block A error bits	See Table 4-164 (page 313)
3:2	Block B error bits	
5:4	Block C error bits	
7:6	Block D error bits	

Table 4-164 Block error bit values

Bit values	Description
0,0	No errors
0,1	1 to 2 errors; slight chance of uncorrected errors
1,0	3 to 5 errors; may have uncorrected errors
1,1	6 or more errors; uncorrectable block

4.7.35 Command 0x1E: GetRdsNotifyMask

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to get an RF tuner accessory's current RDS/RBDS data notification mask (`rdsMask`). In response, the accessory must send a `RetRdsNotifyMask` command with the RDS/RBDS data notification mask. This command is valid only if the RDS/RBDS data-ready capability bit was set in a previous `RetTunerCaps` command.

Table 4-165 `GetRdsNotifyMask` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.7.36 Command 0x1F: RetRdsNotifyMask

Lingo: 0x07 — Origin: Accessory

This command is sent by an RF tuner accessory in response to a `GetRdsNotifyMask` command sent by an Apple device. The RDS/RBDS mask (`rdsMask`) indicates which asynchronous data notifications should be sent by the accessory when data is ready. For each data type bit, 1 means that data-ready notification is enabled and 0 means that notification is disabled. For enabled RDS/RBDS data types, the accessory must send `RdsReadyNotify` commands to the Apple device when the associated data becomes available.

Table 4-166 `RetRdsNotifyMask` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	RDS/RBDS data change notification mask (see Table 4-167 (page 314) and Table 4-168 (page 314))
Packet footer specified in "Command Packets" (page 109).		

Table 4-167 RDS/RBDS data change notification mask bits, parsed mode

Bit	Description
03-00	Reserved
04	RadioText (RT)
29-05	Reserved (must be set to zeros)
30	Program Service Name (PSN)
31	Reserved (must be set to 0)

Table 4-168 RDS/RBDS data change notification mask bits, raw mode

Bit	Description
04-00	Reserved

Bit	Description
05	RDS/RBDS group data
31-06	Reserved

4.7.37 Command 0x20: SetRdsNotifyMask

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to an RF tuner accessory to set the accessory's RDS/RBDS data-ready notification mask (`rdsMask`). When notification bits are enabled, the accessory must send an `RdsReadyNotify` command with the associated RDS/RBDS data. For each RDS/RBDS data type bit, 1 enables data-ready notification and 0 disables notification.

Note: For RDS/RBDS data-ready changes, the `SetRdsNotifyMask` command can override the notification status mask bit. If the `SetRdsNotifyMask` notification mask is set with a nonzero mask, then any enabled RDS/RBDS data notifications will be sent using the `RdsReadyNotify` command instead of the `StatusChangeNotify` command.

Table 4-169 `SetRdsNotifyMask` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	RDS/RBDS data change notification mask (see Table 4-170 (page 315) and Table 4-171 (page 316))
Packet footer specified in "Command Packets" (page 109).		

Table 4-170 RDS/RBDS data change notification mask setting bits, parsed mode

Bit	Description
03-00	Reserved
04	RadioText (RT)
29-05	Reserved
30	Program Service Name (PSN)
31	Reserved

Table 4-171 RDS/RBDS data change notification mask setting bits, raw mode

Bit	Description
04-00	Reserved
05	RDS/RBDS group data
31-06	Reserved

4.7.38 Command 0x21: RdsReadyNotify

Lingo: 0x07 — Origin: Accessory

This command must be sent by an RF tuner accessory when RDS/RBDS data is ready, the associated `SetRdsNotifyMask` bit is set, and the accessory's capability bit from a previous `RetTunerCaps` command indicates that the accessory supports RDS/RBDS status notifications. After a notification command is sent, the accessory's associated RDS/RBDS data-ready status bit must cleared.

Table 4-172 `RdsReadyNotify` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`rdsDataType` (see Table 4-160 (page 312))
0x*NN*	*NN*	`rdsData` (see Table 4-160 (page 312))
Packet footer specified in "Command Packets" (page 109).		

4.7.39 Command 0x25: GetHDProgramServiceCount

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to an RF Tuner accessory to get the count of HD program services broadcast at the current tuner frequency. In response, the accessory must send a `RetHDProgramServiceCount` command with the count of HD program services available. This command is only valid if the HD capable bit was set in a previous `RetTunerCaps` command.

Table 4-173 `GetHDProgramServiceCount` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.7.40 Command 0x26: RetHDProgramServiceCount

Lingo: 0x07 — Origin: Accessory

This command is sent by an RF Tuner accessory in response to a `GetHDProgramServiceCount` command sent by an Apple device. The command returns the count of HD program services broadcast at the current tuner frequency. The command returns a count of 0 if there are no HD program services available. HD programs services must be indexed from 1 to the count value, 1 being the main program service. The Analog Program Exists field indicates whether the station has analog programming or not. This command is only valid if the HD capable bit was set in a previous `RetTunerCaps` command.

Table 4-174 `RetHDProgramServiceCount` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	HD program service count (0-8)
0x*NN*	1	Analog Program Exists (0 = no, 1 = yes)
Packet footer specified in "Command Packets" (page 109).		

4.7.41 Command 0x27: GetHDProgramService

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to an RF Tuner accessory to get the current tuned HD program service. In response, the accessory must send a `RetHDProgramService` command with the current HD program service. This command is only valid if the HD capable bit was set in a previous `RetTunerCaps` command.

Table 4-175 `GetHDProgramService` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.7.42 Command 0x28: RetHDProgramService

Lingo: 0x07 — Origin: Accessory

This command is sent by an RF Tuner accessory in response to a `GetHDProgramService` command sent by an Apple device. The value returned is the tuned HD program service, 0 if the analog program is currently tuned, or 0xFF if audio decoding and output is currently disabled. The output may be disabled if a

SetHDProgramService command was sent with parameter 0xFF or if the radio has just been tuned to a station's frequency. This command is only valid if the HD capable bit was set in a previous RetTunerCaps command.

Table 4-176 RetHDProgramService packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	HD program service index (0x00-0x08 or 0xFF)
Packet footer specified in "Command Packets" (page 109).		

4.7.43 Command 0x29: SetHDProgramService

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to an RF Tuner accessory to tune to an HD program service. Tuning to program service 0 disables HD tuning and switches back to analog tuning, if it is available. Tuning to program service 0xFF disables all audio decoding and output. This command lets the Apple device retrieve information for all available HD programs before selecting a program's audio to be decoded and presented. This command is only valid if the HD capable bit was set in a previous RetTunerCaps command.

Selecting an analog program on an HD digital-only station or attempting to select a nonexistent HD program constitutes an invalid program selection. In this case, the accessory should send the Apple device an AccessoryAck command with Bad Parameter status.

Table 4-177 SetHDProgramService packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	HD program service index (0x00-0x08 or 0xFF)
Packet footer specified in "Command Packets" (page 109).		

4.7.44 Command 0x2A: GetHDDataReadyStatus

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to an RF tuner accessory to get the accessory's current HD data-ready status. It can be used to poll the accessory's HD data-ready status without having to enable HD data-ready notifications. In response, the accessory must send a RetHDDataReadyStatus command. This command is only valid if the HD capable bit was set in a previous RetTunerCaps command.

Note: Only parsed mode is supported for HD Data.

Table 4-178 `GetHDDataReadyStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.7.45 Command 0x2B: RetHDDataReadyStatus

Lingo: 0x07 — Origin: Accessory

This command must be sent by an RF tuner accessory in response to a `GetHDDataReadyStatus` command received from an Apple device. Its status value indicates which HD data values are available to be read. All status bits must remain set on the accessory until the Apple device sends a `GetHDData` command to read the data.

Table 4-179 `RetHDDataReadyStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	HD data-ready status (see Table 4-180 (page 319))
Packet footer specified in "Command Packets" (page 109).		

Table 4-180 HD data-ready status bits

Bit	Description
0	PSD data ready
1	Reserved
2	SIS Station ID number data-ready
3	SIS Station Name (short) data-ready
4	SIS Station Name (long) data-ready
5	SIS ALFN data-ready
6	SIS Station Location data-ready
7	SIS Station Message data-ready

Bit	Description
8	SIS Slogan data-ready
9	SIS Parameter Message data-ready
31:10	Reserved

4.7.46 Command 0x2C: GetHDData

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to get HD data from an RF tuner accessory. If the accessory supports HD and has data-ready, it must send a `RetHDData` command. If the accessory does not support HD or does not have the specified data type ready, it must return an `AccessoryAck` command with command failure status. This command is valid only if the HD capable bit was set in a previous `RetTunerCaps` command.

Note: Only parsed mode is supported for HD Data.

Table 4-181 `GetHDData` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	HD data type; see Table 4-182 (page 320)
Packet footer specified in "Command Packets" (page 109).		

Table 4-182 HD data type IDs

Type ID	Description
0x00	PSD data
0x01	Reserved
0x02	SIS Station ID number data
0x03	SIS Station Name (short) data
0x04	SIS Station Name (long) data
0x05	SIS ALFN data
0x06	SIS Station Location data
0x07	SIS Station Message data
0x08	SIS Slogan

Type ID	Description
0x09	SIS Parameter Message data
0x0A-0xFF	Reserved

4.7.47 Command 0x2D: RetHDData

Lingo: 0x07 — Origin: Accessory

This command is sent by an RF Tuner accessory that has HD data ready in response to a `GetHDData` command. If the accessory does not have the specified data type ready, it must return an `AccessoryAck` command with command failure status (command status 0x02). On sending this command, the accessory must keep clear its internal HD data-ready status for the data type being read until the next time data is ready.

Table 4-183 `RetHDData` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`HDDataType` (see Table 4-184 (page 321))
0x*NN*	*NN*	`HDData` (see Table 4-184 (page 321))
Packet footer specified in "Command Packets" (page 109).		

Table 4-184 `HDDataType` type IDs and formats

HDDataType ID	HDData format
0x00	8 Bit integer index, 8 Bit integer max index, 8 bit HD program index, ID3 Frame data (see Table 4-185 (page 322))
0x01	N/A
0x02	32-bit integer station ID
0x03	8-bit character encoding type, up to 12 bytes (4 or 7 in USA) of short station name (see Table 4-186 (page 322))
0x04	0-56 characters station name (UTF-7)
0x05	32-bit ALFN
0x06	27-bit GPS latitude/longitude encoded in lower bits of 32 bit data
0x07	8-bit character encoding type, 0-190 bytes station message (see Table 4-186 (page 322))
0x08	8-bit character encoding type, 2-96 bytes station slogan (see Table 4-186 (page 322))
0x09	6-bit index encoded in lower bits of 8 bit data, 16-bit parameter

HDDataType ID	HDData format
0x0A–0xFF	N/A

Table 4-185 PSD data format

HDDataType ID	Description
0x00	Current ID3 Frame index (0 = first)
0x01	Last Index of ID3 Frame data
0x02	HD Program Index (1-8)
0x03–0x*NN*	ID3 Frame data, ID3 tag version 2.3.0. The first byte of the ID3 tag's text information contains the encoding type.

Table 4-186 8-bit character encoding type formats

Encoding type ID	Description
0x00	ISO/IEC 8859-1:1998
0x01–0x03	Reserved
0x04	ISO/IEC 10646-1:2000, UCS-2 (Little-endian)
0x05–0xFF	Reserved

4.7.48 Command 0x2E: GetHDDataNotifyMask

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to get an RF tuner accessory's current HD data notification mask. In response, the accessory must send a `RetHDDataNotifyMask` command with the HD data notification mask. This command is valid only if the HD capability bit was set in a previous `RetTunerCaps` command.

Table 4-187 `GetHDDataNotifyMask` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.7.49 Command 0x2F: RetHDDataNotifyMask

Lingo: 0x07 — Origin: Accessory

This command is sent by an RF tuner accessory in response to a `GetHDDataNotifyMask` command sent by an Apple device. The HD mask indicates which asynchronous data notifications should be sent by the accessory when data is ready. For each data type bit, 1 means that data-ready notification is enabled and 0 means that notification is disabled. For enabled HD data types, the accessory must send `HDDataReadyNotify` commands to the Apple device when the associated data becomes available.

Table 4-188 `RetHDDataNotifyMask` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	HD data change notification mask (see Table 4-189 (page 323))
Packet footer specified in "Command Packets" (page 109).		

Table 4-189 HD data change notification mask bits

Bit	Description
0	PSD data ready
1	Reserved
2	SIS Station ID number data-ready
3	SIS Station Name (short) data-ready
4	SIS Station Name (long) data-ready
5	SIS ALFN data-ready
6	SIS Station Location data-ready
7	SIS Station Message data-ready
8	SIS Slogan data-ready
9	SIS Parameter Message data-ready
31:10	Reserved

4.7.50 Command 0x30: SetHDDataNotifyMask

Lingo: 0x07 — Origin: Apple device

This command is sent by an Apple device to an RF tuner accessory to set the accessory's HD data-ready notification mask. When notification bits are enabled, the accessory must send an `HDDataReadyNotify` command with the associated HD data. For each HD data type bit, 1 enables data-ready notification and 0 disables notification. This command is valid only if the HD capability bit was set in a previous `RetTunerCaps` command.

Table 4-190 `SetHDDataNotifyMask` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	HD data change notification mask (see Table 4-191 (page 324))
Packet footer specified in "Command Packets" (page 109).		

Table 4-191 HD data change notification mask setting bits

Bit	Description
0	PSD data ready
1	Reserved
2	SIS Station ID number data-ready
3	SIS Station Name (short) data-ready
4	SIS Station Name (long) data-ready
5	SIS ALFN data-ready
6	SIS Station Location data-ready
7	SIS Station Message data-ready
8	SIS Slogan data-ready
9	SIS Parameter Message data-ready
31:10	Reserved

4.7.51 Command 0x31: HDDataReadyNotify

Lingo: 0x07 — Origin: Accessory

This command must be sent by an RF tuner accessory when HD data is ready, the associated `SetHDDataNotifyMask` bit is set, and the accessory's capability bit from a previous `RetTunerCaps` command indicates that the accessory supports HD status notifications. After a notification command is sent, the accessory's associated HD data-ready status bit must cleared.

Table 4-192 `HDDataReadyNotify` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`HDDataType` (see Table 4-193 (page 325))
0x*NN*	*NN*	HD data (see Table 4-194 (page 325))
Packet footer specified in "Command Packets" (page 109).		

Table 4-193 HD data types

Type ID	Description
0x00	PSD data
0x01	Reserved
0x02	SIS Station ID number data
0x03	SIS Station Name (short) data
0x04	SIS Station Name (long) data
0x05	SIS ALFN data
0x06	SIS Station Location data
0x07	SIS Station Message data
0x08	SIS Slogan data
0x09	SIS Parameter Message data
0x0A–0xFF	Reserved

Table 4-194 HD Data Type type IDs and formats

HD data type ID	HDData format
0x00	8 Bit integer index, 8 Bit integer max index, 8 bit HD program index, ID3 Frame data (see Table 4-195 (page 326))
0x01	N/A
0x02	32-bit integer station ID
0x03	8-bit character encoding type, up to 12 bytes (4 or 7 in USA) of short station name (see Table 4-196 (page 326))
0x04	0-56 characters station name (UTF-7)

HD data type ID	HDData format
0x05	32-bit ALFN
0x06	27-bit GPS latitude/longitude encoded in lower bits of 32 bit data
0x07	8-bit character encoding type, 0-190 bytes station message (see Table 4-196 (page 326))
0x08	8-bit character encoding type, 2-96 bytes station slogan (see Table 4-196 (page 326))
0x09	6-bit index encoded in lower bits of 8 bit data, 16-bit parameter
0x0A–0xFF	N/A

Table 4-195 PSD data format

HDDataType ID	Description
0x00	Current ID3 Frame index (0 = first)
0x01	Last Index of ID3 Frame data
0x02	HD Program Index (1-8)
0x03–0x*NN*	ID3 Frame data, ID3 tag version 2.3.0. The first byte of the ID3 tag's text information contains the encoding type.

Table 4-196 8-bit character encoding type formats

Encoding type ID	Description
0x00	ISO/IEC 8859-1:1998
0x01–0x03	Reserved
0x04	ISO/IEC 10646-1:2000, UCS-2 (Little-endian)
0x05–0xFF	Reserved

4.7.52 Sample HD Command Sequences

This section presents three typical examples of using RF Tuner lingo commands with HD radio.

4.7.52.1 Initializing HD Radio

This example sets up HD Radio using RF Tuner lingo commands.

Table 4-197 Example of HD radio setup

Command	Direction	Data	Comments
GetTunerCaps	Dev to Acc	No Data	Request RF Tuner accessory's supported capabilities.
RetTunerCaps	Acc to Dev	0x07FC0712	Return 32-bit field indicating the accessory's capabilities (FM band US, HD Radio, Tuner power control, status change notification, FM resolution 200 kHz, Tuner seek capable, Tuner seek RSSI threshold capable, Force monophonic mode capable, Stereo blend capable, FM Tuner deemphasis select capable, AM resolution 10 kHz, RDS/RBDS data capable, Tuner channel RSSI indication capable, Stereo source indicator capable).
SetTunerCtrl	Dev to Acc	0x01	Tuner power on, status change notification off, raw mode off.
AccessoryAck	Acc to Dev	0x00	Acknowledge SetTunerCtrl
SetStatusNotifyMask	Dev to Acc	0x38	Set status notification for HD signal present, HD digital audio present, HD data ready.
AccessoryAck	Acc to Dev	0x00	Acknowledge SetStatusNotifyMask
SetHDDataNotifyMask	Dev to Acc	0x00000009	Set HD data ready notification for Station Short Name and PSD Data.
AccessoryAck	Acc to Dev	0x00	Acknowledge SetHDDataNotifyMask
SetTunerMode	Dev to Acc	0x00	Set FM Tuner resolution 200kHz, stereo allowed, no stereo blend, FM Tuner de-emphasis 75 µsec, AM Tuner resolution 10kHz.
AccessoryAck	Acc to Dev	0x00	Acknowledge SetTunerMode

4.7.52.2 HD Radio Tuning and Reception

Using RF Tuner lingo commands, this example tunes to an HD radio station, collects data on it, and responds to a loss of digital audio.

Table 4-198 Example of HD radio tuning and reception

Command	Direction	Data	Comments
SetTunerBand	Dev to Acc	0x01	Set Tuner band to FM US.
AccessoryAck	Acc to Dev	0x00	Acknowledge SetTunerBand
SetTunerFreq	Dev to Acc	97700	Set FM Tuner frequency to 97.7 MHz
RetTunerFreq	Acc to Dev	97700, 31	Return FM Tuner frequency set to 97.7 MHz with RSSI level of 31.

Command	Direction	Data	Comments
SetTunerSeekRssi	Dev to Acc	8	Set the tuner seek RSSI threshold to 8.
AccessoryAck	Acc to Dev	0x00	Acknowledge SetTunerSeekRssi
TunerSeekStart	Dev to Acc	0x0F	Start tuner seek up from current frequency while checking for RSSI threshold.
AccessoryAck	Acc to Dev	0x00	Acknowledge TunerSeekStart
TunerSeekDone	Acc to Dev	106900, 30	Tuner seek has finished with tuned frequency of 106.9 MHz and RSSI level of 30.
GetHDProgramService-Count	Dev to Acc	No data	Retrieve number of HD Program Services available.
RetHDProgramService-Count	Acc to Dev	3,1	3 HD Program services and the analog programming are available.
SetHDProgramService	Dev to Acc	1	Start decode and playback of HD program service #1.
AccessoryAck	Acc to Dev	0x00	Acknowledge SetHDProgramService
Collect info on HD programs, as shown in "HD Radio Service Management" (page 329). Info collection could have been performed before tuning by using SetHDDataNotifyMask; see "Initializing HD Radio" (page 326).			
Set notification:			
SetStatusNotifyMask	Dev to Acc	0x18	Set status notification for HD signal present, HD digital audio present.
AccessoryAck	Acc to Dev	0x00	Acknowledge SetStatusNotifyMask
Retrieve HD signal and audio status:			
StatusChangeNotify	Acc to Dev	0x18	Status change notification: HD signal and HD audio present.
GetHDProgramService-Count	Dev to Acc	No data	Retrieve number of HD Program Services available.
RetHDProgramService-Count	Acc to Dev	2,1	2 HD Program services and the analog programming are available.
SetHDProgramService	Dev to Acc	1	Start decode/playback of HD program service #1.
AccessoryAck	Acc to Dev	0x00	Acknowledge SetHDProgramService
Respond to loss of HD digital audio:			

Command	Direction	Data	Comments
StatusChangeNotify	Acc to Dev	0x08	Status change notification: HD Signal present, HD digital audio and data not present.
GetHDProgramService-Count	Dev to Acc	No data	Retrieve number of HD Program Services available.
RetHDProgramService-Count	Acc to Dev	0,1	Zero HD Program services and the analog programming are available.
SetHDProgramService	Dev to Acc	0	Start playback of analog programming.
AccessoryAck	Acc to Dev	0x00	Acknowledge SetHDProgramService

4.7.52.3 HD Radio Service Management

This example retrieves PSD and other data for HD radio, using RF Tuner lingo commands.

Table 4-199 Example of getting PSD and name data

Command	Direction	Data	Comments
SetHDDataNotifyMask	Dev to Acc	0x00000009	Set HD data ready notification for Station Short Name and PSD Data.
AccessoryAck	Acc to Dev	0x00	Acknowledge SetHDDataNotifyMask
HDDataReadyNotify	Acc to Dev	0x03, 1, NNNN	Station Short Name received, ISO 8859-1.
HDDataReadyNotify	Acc to Dev	0x00, 0, 1, 1, NNNN	PSD ID3 Frame data 1 of 2 received for program 1.
HDDataReadyNotify	Acc to Dev	0x00, 1, 1, 1, NNNN	PSD ID3 Frame data 2 of 2 received for program 1.
HDDataReadyNotify	Acc to Dev	0x00, 0, 0, 2, NNNN	PSD ID3 Frame data 1 of 1 received for program 2.
GetHDDataReadyStatus	Dev to Acc	No Data	Ask accessory to return HD data ready status.
RetHDDataReadyStatus	Acc to Dev	0x00000009	Station Short Name and PSD data available.
GetHDData	Dev to Acc	0x03	Ask accessory for Station Short Name.
RetHDData	Acc to Dev	0x03, 1, NNNN	Station Short Name, ISO 8859-1.
GetHDData	Dev to Acc	0x00	Ask accessory for PSD Data.
RetHDData	Acc to Dev	0x00, 0, 1, 1, NNNN	PSD ID3 Frame data 1 of 2 received for program 1.

Command	Direction	Data	Comments
RetHDData	Acc to Dev	0x00, 1, 1, 1, NNNN	PSD ID3 Frame data 2 of 2 received for program 1.
GetHDDataReadyStatus	Dev to Acc	No Data	Ask accessory to return HD data ready status.
RetHDDataReadyStatus	Acc to Dev	0x00000001	PSD data available.
GetHDData	Dev to Acc	0x00	Ask accessory for PSD Data.
RetHDData	Acc to Dev	0x00, 0, 0, 2, NNNN	PSD ID3 Frame data 1 of 1 received for program 2.
GetHDDataReadyStatus	Dev to Acc	No Data	Ask accessory to return HD data ready status.
RetHDDataReadyStatus	Acc to Dev	0x00000000	No more data available.

4.8 Lingo 0x08: Accessory Equalizer Lingo

Apple devices may be connected to accessory devices such as boom boxes or amplifiers that are capable of frequency response equalization. The Accessory Equalizer lingo, number 0x08, lets the Apple device control the Equalizer Settings of the accessory.

Note: The Accessory Equalizer lingo requires accessory authentication for all communication transports (serial or USB).

The Accessory Equalizer lingo is similar to the Display Remote Equalizer lingo, but the command direction is reversed: the Apple device is the master, initiating commands to which the accessory responds. The Apple device can query the total count of accessory Equalizer Settings and the UTF-8 name for each one; it can then get or set each Equalizer Setting on the accessory.

To support this lingo, the Apple device's application software will add an Apple device menu item for accessory equalizer selection and control.

4.8.1 Equalizer Setting Requirements

To be compatible with the Accessory Equalizer lingo, an accessory must observe these rules:

- The accessory must have an Equalizer Setting with an index of 0x00 and this must be the accessory equalizer's off/none setting.
- The Equalizer Index and Equalizer Setting count fields are 1 byte in size, limiting the setting count to a maximum of 255, including the required index 0x00 setting.

- Devices supporting this lingo must support at least two Equalizer Settings: off/none and at least one other setting. If fewer than two settings are returned by the accessory, the Apple device will not present equalizer menu options to the user.

4.8.2 Accessory Equalizer Lingo Commands

Table 4-200 (page 331) lists the commands included in the Accessory Equalizer lingo.

Table 4-200 Accessory Equalizer lingo command summary

Command	ID	Direction	Data length	Protocol version	Authentication required
`AccessoryAck`	0x00	Acc to Dev	4	1.00	Yes
`GetCurrentEQIndex`	0x01	Dev to Acc	2	1.00	Yes
`RetCurrentEQIndex`	0x02	Acc to Dev	3	1.00	Yes
`SetCurrentEQIndex`	0x03	Dev to Acc	3	1.00	Yes
`GetEQSettingCount`	0x04	Dev to Acc	2	1.00	Yes
`RetEQSettingCount`	0x05	Acc to Dev	3	1.00	Yes
`GetEQIndexName`	0x06	Dev to Acc	3	1.00	Yes
`RetEQIndexName`	0x07	Acc to Dev	0x*NN*	1.00	Yes
Reserved	0x08–0xFF	N/A	N/A	N/A	N/A

4.8.2.1 Command History of the Accessory Equalizer Lingo

Table 4-201 (page 331) shows the history of command changes in the accessory equalizer lingo:

Table 4-201 Accessory Equalizer lingo command history

Lingo version	Command changes	Features
1.00	Add: 0x00–0x07	Accessory Equalizer Index, count, and name support

4.8.3 Command 0x00: AccessoryAck

Lingo: 0x08 — Origin: Accessory

This command must be sent by the accessory in response to commands sent from the Apple device. An `AccessoryAck` response must be sent when a bad parameter is received, an unsupported/invalid command is received, or a command completed but did not return any data. See Table 4-202 (page 332).

Table 4-202 `AccessoryAck` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command status; see Table 3-6 (page 125).
0x*NN*	1	ID of the command for which the response is being sent
Packet footer specified in "Command Packets" (page 109).		

4.8.4 Command 0x01: GetCurrentEQIndex

Lingo: 0x08 — Origin: Apple device

This command is sent by the Apple device to request the current Equalizer Setting from the accessory. In response, the accessory sends a `RetCurrentEQIndex` command with the current Equalizer Index.

The timeout for this command is 2500 ms (2.5 seconds). If the accessory does not respond within the allotted time, the Apple device will retry up to three times. If the accessory does not respond within the specified time and retry count, the command will fail.

Table 4-203 `GetCurrentEQIndex` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.8.5 Command 0x02: RetCurrentEQIndex

Lingo: 0x08 — Origin: Accessory

This command is sent by the accessory in response to the `GetCurrentEQIndex` command received from the Apple device. The Equalizer Index returned may range from 0 (reserved for the off/none Equalizer Setting) to 254 (the maximum possible Equalizer Index). Devices are not required to support 255 settings; they may support as few as the two minimum required settings.

Table 4-204 `RetCurrentEQIndex` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`eqIndex`: The current accessory Equalizer Index. See Table 4-205 (page 333).
Packet footer specified in "Command Packets" (page 109).		

Table 4-205 Accessory Equalizer Setting indices

Index	Description
0x00	Equalizer off/none
0x01	First Equalizer Setting
0x*NN*	Last Equalizer Setting (accessory-dependent maximum index)

4.8.6 Command 0x03: SetCurrentEQIndex

Lingo: 0x08 — Origin: Apple device

This command is sent by the Apple device to select an accessory Equalizer Index from the supported range. The valid range is from 0 to one less than the Equalizer Setting count returned by the `RetEQSettingCount` command. See Table 4-205 (page 333).

Note: Accessories may receive duplicate `SetCurrentEQIndex` commands at any time and must handle them correctly.

The timeout for this command is 3000 ms (3.0 seconds). If the accessory does not respond within the allotted time, the Apple device will retry up to three times. If the accessory does not respond within the specified time and retry count, the command will fail.

Table 4-206 `SetCurrentEQIndex` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`eqIndex`: the selected accessory Equalizer Index
Packet footer specified in "Command Packets" (page 109).		

4.8.7 Command 0x04: GetEQSettingCount

Lingo: 0x08 — Origin: Apple device

This command is sent by the Apple device to determine how many Equalizer Settings the accessory supports. In response, the accessory sends a `RetEQSettingCount` command with the count of Equalizer Settings supported.

The timeout for this command is 3000 ms (3.0 seconds). If the accessory does not respond within the allotted time, the Apple device will retry up to three times. If the accessory does not respond within the specified time and retry count, the command will fail.

Table 4-207 `GetEQSettingCount` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.8.8 Command 0x05: RetEQSettingCount

Lingo: 0x08 — Origin: Accessory

This command is sent by the accessory in response to a `GetEQSettingCount` command from the Apple device. To support the Accessory Equalizer lingo, an accessory must report a minimum count of 0x02 (equalizer off/none plus one other setting) and may support a maximum count of 0xFF (equalizer off/none plus 254 other settings).

Table 4-208 `RetEQSettingCount` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`eqCount`: the count of accessory equalizer indices. See Table 4-209 (page 334).
Packet footer specified in "Command Packets" (page 109).		

Table 4-209 `RetEQSettingCount` parameter values

Range	Comment
0x00–0x01	Invalid equalizer count (minimum count is 0x02)
0x02–0xFF	Valid equalizer count range

4.8.9 Command 0x06: GetEQIndexName

Lingo: 0x08 — Origin: Apple device

This command is sent by the Apple device to obtain the name string associated with a specified Equalizer Index. In response, the accessory sends a `RetEQIndexName` command with the same Equalizer Index and the associated Equalizer Setting name string.

The timeout for this command is 3000 ms (3.0 seconds). If the accessory does not respond within the allotted time, the Apple device will retry up to three times. If the accessory does not respond within the specified time and retry count, the command will fail.

Table 4-210 `GetEQIndexName` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`eqIndex`: the selected accessory Equalizer Index
Packet footer specified in "Command Packets" (page 109).		

4.8.10 Command 0x07: RetEQIndexName

Lingo: 0x08 — Origin: Accessory

This command is sent by the accessory in response to a `GetEQIndexName` command received from the Apple device. The `eqIndex` byte is the same index that was sent by the `GetEQIndexName` command. The `eqName` string is a null-terminated UTF-8 character array. The array length must not exceed 32 characters plus a null terminator character. This length limit minimizes truncation of the Equalizer Setting name when it is displayed in the Apple device's accessory Equalizer Settings menu.

Table 4-211 `RetEQIndexName` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`eqIndex`: the selected accessory Equalizer Index
0x*NN*	*NN*	The accessory equalizer name, as a null-terminated UTF-8 character array
Packet footer specified in "Command Packets" (page 109).		

4.9 Lingo 0x09: Sports Lingo

The Sports lingo enables gym cardio equipment to communicate with an Apple device to track a user's workout while the user continues to use the Apple device for other purposes. The Sports lingo version of the Apple device must be 1.01 or later to support these accessories. For information about determining the lingo version, see "Determining Apple Device Support for Cardio Equipment" (page 501).

All Sports lingo commands require authentication level 2.0.

4.9.1 Sports Lingo commands

Table 4-212 (page 336) summarizes the Sports commands (lingo 0x09).

Table 4-212 Sports lingo command summary

Command	ID	Direction	Data length	Protocol version
`AccessoryAck`	0x00	Acc to Dev	0x02	1.01
`GetAccessoryVersion`	0x01	Dev to Acc	0x00	1.01
`RetAccessoryVersion`	0x02	Acc to Dev	0x02	1.01
`GetAccessoryCaps`	0x03	Dev to Acc	0x00	1.01
`RetAccessoryCaps`	0x04	Acc to Dev	0x03	1.01
Reserved	0x05-0x7F	N/A		
`iPodAck`	0x80	Dev to Acc	0x02	1.01
Reserved	0x81-0x82	N/A		
`GetiPodCaps`	0x83	Acc to Dev	0x00	1.01
`RetiPodCaps`	0x84	Dev to Acc	0x03	1.01
`GetUserIndex`	0x85	Acc to Dev	0x00	1.01
`RetUserIndex`	0x86	Dev to Acc	0x01	1.01
Reserved	0x87	N/A		
`GetUserData`	0x88	Acc to Dev	0x01	1.01
`RetUserData`	0x89	Dev to Acc	0x*NN*	1.01
`SetUserData`	0x8A	Acc to Dev	0x*NN*	1.01
Reserved	0x8B-0xFF	N/A		

4.9.2 Command History of the Sports Lingo

Table 4-213 (page 336) shows the history of changes to the Sports lingo.

Table 4-213 Sports lingo command history

Lingo version	Command changes	Features
1.01	Add: 0x00–0x04, 0x80, 0x83-0x86, 0x88-0x8A	Cardio equipment accessory support

4.9.3 Command 0x00: AccessoryAck

Lingo: 0x09 — Origin: Accessory

This command is sent by the accessory in response to a command sent from the Apple device. An AccessoryAck response must be sent for any of the following conditions:

- A bad parameter is received
- An unsupported/invalid command is received
- A command that does not return any data has completed

Table 4-214 `AccessoryAck` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command status; see Table 3-6 (page 125).
0x*NN*	1	ID of command being acknowledged
Packet footer specified in "Command Packets" (page 109).		

4.9.4 Command 0x01: GetAccessoryVersion

Lingo: 0x09 — Origin: Apple device

This command is sent by Apple device to obtain the sports accessory lingo protocol version. In response, the accessory will send the `RetAccessoryVersion` command with its major and minor protocol version numbers. Devices may query the Apple device's Sports Lingo protocol version using the General Lingo `RequestLingoProtocolVersion` command.

Table 4-215 `GetAccessoryVersion` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.9.5 Command 0x02: RetAccessoryVersion

Lingo: 0x09 — Origin: Accessory

This command must be returned by the accessory in response to each `GetAccessoryVersion` command received from the Apple device. The accessory returns the maximum lingo version that it supports. The first byte is the major version number (tens/ones digits left of the decimal point) and the second byte is the minor version number (tenths/hundredths digits right of the decimal point). The Apple device will support all accessory lingo versions up to and including its own current lingo version.

Table 4-216 `RetAccessoryVersion` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Major Sports lingo protocol version supported by the accessory (currently 0x01)
0x*NN*	1	Minor Sports lingo protocol version supported by the accessory (currently 0x01)
Packet footer specified in "Command Packets" (page 109).		

4.9.6 Command 0x03: GetAccessoryCaps

Lingo: 0x09 — Origin: Apple device

This command is sent by the Apple device to get the sports accessory capabilities and determine the features present on the accessory. In response, the accessory will send the `GetAccessoryCaps` command with the payload indicating the capabilities supported.

Table 4-217 `GetAccessoryCaps` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.9.7 Command 0x04: RetAccessoryCaps

Lingo: 0x09 — Origin: Accessory

This command must be sent by the accessory in response to each `GetAccessoryCaps` command sent by the Apple device. The sports accessory returns the payload indicating which of the capabilities it supports. When a capability bit is set, it means that the associated command is supported by the accessory. Conversely, if a capability bit is clear, it means that the associated commands are not supported by the accessory.

Table 4-218 `RetAccessoryCaps` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	2	`capsMask`; see Table 4-219 (page 339).
0x00	1	Reserved (set equal to 0x00)
Packet footer specified in "Command Packets" (page 109).		

Table 4-219 `RetAccessoryCaps` `capsMask` values

Value	Description
bits 0-8	Reserved (set to 0)
bit 9	Accessory supports (1) or does not support (0) Sports lingo commands 0x80 through 0x8A
bits 15:10	Reserved (set to 0)

4.9.8 Command 0x80: iPodAck

Lingo: 0x09 — Origin: Apple device

This command is sent by the Apple device in response to a command sent from the accessory. An `iPodAck` response is sent when a bad parameter is received, an unsupported/invalid command is received, or a command that does not return any data has completed.

Table 4-220 `iPodAck` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command status; see Table 3-6 (page 125).
0x*NN*	1	ID of Command being acknowledged
Packet footer specified in "Command Packets" (page 109).		

4.9.9 Command 0x83: GetiPodCaps

Lingo: 0x09 — Origin: Accessory

This command may be sent by the accessory to get the Apple device capabilities and determine if required features are present. In response, the Apple device will send the `RetiPodCaps` command with the payload indicating the capabilities supported.

Note: The `GetiPodCaps` command must be sent before the accessory may use any commands that rely upon Apple device support for gym cardio equipment features, such as `GetUserIndex`.

Table 4-221 `GetiPodCaps` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		

Value	Bytes	Parameter
Packet footer specified in "Command Packets" (page 109).		

4.9.10 Command 0x84: RetiPodCaps

Lingo: 0x09 — Origin: Apple device

This command is sent by the Apple device in response to the `GetiPodCaps` command sent by the accessory. The Apple device returns the payload indicating which of the capabilities it supports. The user count is valid only if the Apple device user data capability bit is set.

Table 4-222 `RetiPodCaps` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x00	2	`capsMask` ; see Table 4-223 (page 340).
0x*NN*	1	`userCount` number of user data profiles on the Apple device
Packet footer specified in "Command Packets" (page 109).		

Table 4-223 `RetiPodCaps capsMask` details

capsMask	Description
bit 0	Cardio equipment is supported (1) or is not supported (0).
bit 1	User data is supported (1) or is not supported (0). The `userCount` value is valid if and only if this capability bit is set.
bits 15:2	Reserved (set to 0)

4.9.11 Command 0x85: GetUserIndex

Lingo: 0x09 — Origin: Accessory

This command may be sent by the accessory to obtain the current user index selection from the Apple device. In response, the Apple device will return the `RetUserIndex` command with the current user index.

Note: This command must not be sent unless the accessory has already sent a `GetiPodCaps` command and received a `RetiPodCaps` command indicating user data support capability. If the Apple device does not support user data, it will return an `iPodAck` with a bad parameter status.

Table 4-224 `GetUserIndex` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.9.12 Command 0x86: RetUserIndex

Lingo: 0x09 — Origin: Apple device

This command is sent by the Apple device in response to the `GetUserIndex` command received from the accessory. The current user index is returned (the first user has index 0). The total user count present on the Apple device is obtained using the `GetiPodCaps` command.

Table 4-225 `RetUserIndex` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`userIndex` (0 to `userCount`-1) (see Note below)
Packet footer specified in "Command Packets" (page 109).		

Note: The `userIndex` value 0 is reserved by the Apple device for the default/unknown user profile. An Apple device that does not support multiple user profiles but does support user data will offer only `userIndex` 0.

4.9.13 Command 0x88: GetUserData

Lingo: 0x09 — Origin: Accessory

This command may be sent by the accessory to obtain the current user's data. In response, the Apple device will return the `RetUserData` command with the specified `userDataType`. If the Apple device does not support this command or the `userDataType` is not valid, the Apple device will return an `iPodAck` with a bad parameter status.

Note: This command must not be sent unless the accessory has already sent a `GetiPodCaps` command and received a `RetiPodCaps` command indicating user data support capability. If the Apple device does not support user data, it will return an `iPodAck` with a bad parameter status.

Table 4-226 `GetUserData` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`userDataType`; see Table 4-227 (page 342).
Packet footer specified in "Command Packets" (page 109).		

Table 4-227 `GetUserData` `userDataType` values

userDataType	Description
0x00	Preferred Unit System
0x01	Name
0x02	Gender
0x03	Weight
0x04	Age
0x05	Workout Recording Preference
0x06-0xFF	Reserved

4.9.14 Command 0x89: RetUserData

Lingo: 0x09 — Origin: Apple device

This command is sent by the Apple device in response to the `GetUserData` command received from the accessory. The Apple device returns the requested user data for the specified data type.

Table 4-228 `RetUserData` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`userDataType`; see Table 4-229 (page 343).
0x*NN*	*NN*	`userData` (variable length)
Packet footer specified in "Command Packets" (page 109).		

Table 4-229 `RetUserData` `userDataType` details

userDataType	userData
0x00	Preferred unit system (1 byte): 0x00 = No information 0x01 = Imperial units (lbs/miles) 0x02 = Metric units (kg/km) 0x03-0xFF = Reserved
0x01	Name (variable length, 128 bytes max, null-terminated UTF-8 string)
0x02	Gender (1 byte): 0x00 = No information 0x01 = Female 0x02 = Male 0x03-0xFF = Reserved
0x03	Weight in tenths of a kg (2 bytes, MSB first): 0x0000 = No information 0x0001 = 0.1 kg 0x0200 = 51.2 kg 0x1388 = 500.0 kg (maximum weight limit) 0x1389-0xFFFF = Reserved
0x04	Age in years (1 byte): 0x00 = No information 0x20 = 32 years old 0xC8 = 200 years old (maximum age limit) 0xC9-0xFF = Reserved
0x05	Workout recording preference (1 byte): 0x00 = No information 0x01 = Never record workout data 0x02 = Ask if workout should be recorded 0x03 = Always record workout data 0x04-0xFF = Reserved
0x06-0xFF	Reserved

4.9.15 Command 0x8A: SetUserData

This command may be sent by the accessory to set the current user's data. In response, the Apple device will send the `iPodAck` command with the status of the operation. If the Apple device does not support this command or the `userDataType` or `userData` is not valid, an `iPodAck` with a bad parameter status is returned.

Note: This command must not be sent unless the accessory has already sent a `GetiPodCaps` command and received a `RetiPodCaps` command indicating user data support capability. If the Apple device does not support user data, it will return an `iPodAck` with a bad parameter status.

Table 4-230 `SetUserData` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`userDataType`; see Table 4-231 (page 344).
0x*NN*	*NN*	`userData` (variable length)
Packet footer specified in "Command Packets" (page 109).		

Table 4-231 `SetUserData userDataType` details

userDataType	userData
0x00	Preferred unit system (1 byte): 0x00 = No information 0x01 = Imperial units (lbs/miles) 0x02 = Metric units (kg/km) 0x03-0xFF = Reserved
0x01	Setting not allowed
0x02	Gender (1 byte): 0x00 = No information 0x01 = Female 0x02 = Male 0x03-0xFF = Reserved
0x03	Weight in tenths of a kg (2 bytes, MSB first): 0x0000 = No information 0x0001 = 0.1 kg 0x0200 = 51.2 kg 0x1388 = 500.0 kg (maximum weight limit) 0x1389-0xFFFF = Reserved

userDataType	userData
0x04	Age in years (1 byte): 0x00 = No information 0x20 = 32 years old 0xC8 = 200 years old (maximum age limit) 0xC9-0xFF = Reserved
0x05	Setting not allowed
0x06-0xFF	Reserved

4.10 Lingo 0x0A: Digital Audio Lingo

Digital Audio lingo commands are used to configure USB Device Mode audio on Apple devices which support that feature (see "General Apple Device Features" (page 44)). The Apple device uses these lingo commands to retrieve a list of supported sample rates from the accessory and to inform the accessory of the Apple device's current sample rate, Sound Check value, and track volume adjustment value (which is set using iTunes). After these interchanges, the Apple device performs audio sample rate conversion internally and transfers digital audio to the accessory at one of the accessory's supported audio data sample rates. For a sample command sequence that declares this lingo, see Table 4-295 (page 393).

Note: The Apple devices that contain version 1.00 of the Digital Audio lingo do not correctly support digital audio. An accessory should check the attached Apple device's version of the Digital Audio lingo and use digital audio only if the version number is greater than 1.00. Version 1.01 of Digital Audio may be used only while the Apple device is in Extended Interface mode (Lingo 0x04); later versions do not have this restriction. See Table 1-2 (page 42) and Table 1-3 (page 42) for lists of affected Apple device models.

4.10.1 Accessory Authentication

Every accessory that supports the Digital Audio lingo must authenticate itself with a connected Apple device as soon as the Apple device recognizes the accessory; deferred authentication is not permitted. This means that the accessory's IDPS process must specify authentication immediately after identification in its `IdentifyToken` (see Table 3-69 (page 163)). With Apple devices that do not support IDPS, the accessory must send an `IdentifyDeviceLingoes` command with the authentication control bits in its `options` field set to 10_b (Authenticate immediately after identification). See Table 3-25 (page 135) for details.

Note: The General lingo `Identify` command cannot be used to identify an accessory for any purpose if the accessory supports the Digital Audio lingo.

When the accessory identifies itself as supporting the Digital Audio lingo, authentication can happen in the background and the Apple device can proceed to transfer digital audio as if authentication were successful.

4.10.2 Digital Audio Lingo Commands

The Digital Audio lingo ID is 0x0A. All its commands are transferred in small packet format and all require authentication. They are listed in Table 4-232 (page 346).

Table 4-232 Digital Audio lingo command summary

ID	Command	Data length	Protocol version	Authentication required
0x00	`AccessoryAck`	0x04	1.00	Yes
0x01	`iPodAck`	0x04	1.00	Yes
0x02	`GetAccessorySampleRateCaps`	0x02	1.00	Yes
0x03	`RetAccessorySampleRateCaps`	*NN*	1.00	Yes
0x04	`TrackNewAudioAttributes`	0x0E	1.00	Yes
0x05	`SetVideoDelay`	0x06	1.03	Yes
0x06–0xFF	`Reserved`	N/A	N/A	N/A

4.10.3 Command History of the Digital Audio Lingo

Table 4-233 (page 346) shows the history of command changes in the Digital Audio lingo:

Table 4-233 Digital Audio lingo command history

Lingo version	Command changes	Features
1.00	Add: 0x00–0x04	Digital audio sample rate and track information support
1.01	None	The `TrackNewAudioAttributes` command is resent until it is acknowledged by the USB host with an `AccessoryAck` command. Also corrected a bug where `TrackNewAudioAttributes` was not being sent before every track; for this reason, accessories should not use Digital Audio with Apple devices that support only Digital Audio lingo version 1.00.
1.02	None	The Digital Audio lingo no longer requires the Apple device to be in Extended Interface mode.
1.03	Add 0x05	Added `SetVideoDelay` to allow digital audio to synchronize with video playback.

4.10.4 USB Device Mode Audio

When an Apple device is in USB Device mode, it can stream digital audio to an accessory as a USB Audio 1.0 peripheral. The audio stream contains PCM data and is synchronized using USB 1 ms start-of-frame (SOF) packets. The Digital Audio lingo must be used to drive this process. For transport details, see "Using an Apple Device as a USB Device" in *MFi Accessory Hardware Specification*.

Note: The Apple device USB isochronous audio data endpoint descriptor `bmAttributes` field erroneously returns the Synchronization Type field (D3:2) as b00 (no synchronization) instead of the correct value, b11 (synchronous). The Apple device supports synchronous data transfers, so USB host devices must override these attribute bits. The erroneous b00 value is retained for backwards compatibility with older accessories.

Note: Any accessory that uses USB Device Mode Audio must comply with the Accessory Design Requirements section of *MFi Accessory Hardware Specification*.

When receiving USB Device mode audio over a USB streaming interface, the accessory may buffer data to achieve consistent audio playback. Since buffering introduces latency, it is suggested that this latency be kept below 200 ms to ensure timely response to user input. Apple may enforce this suggestion in the future.

4.10.4.1 Digital Audio and Extended Interface Mode

Digital Audio lingo version 1.01 requires the Apple device to be in Extended Interface mode before USB Device mode Audio can be transferred. With versions 1.02 and later, USB Device mode Audio can be transferred in any mode.

4.10.4.2 Audio/Video Synchronization

Digital Audio lingo versions 1.03 and above allow synchronization between Apple device video and the sound for that video that is being transmitted as digital audio. The video may either appear on the Apple device's display or be transmitted through the analog video output.

Accessories that support both video output and digital audio streaming from an Apple device must maintain synchronized digital audio/video playback whether or not the Apple device supports the Digital Audio lingo `SetVideoDelay` command. To do this, the accessory must first determine the Apple device's audio routing support, after which the accessory can select the best way to maintain digital audio/video synchronization. To achieve the best available Apple device audio routing, the accessory should do the following:

1. Send a General lingo `StartIDPS` command to begin the accessory identification process.
2. Send a General lingo `GetiPodOptionsForLingo` command for the digital audio lingo (0x0A). If the Apple device returns a `RetiPodOptionsForLingo` command with bit 00 set to 1, continue to Step 3. If the Apple device does not support the `GetiPodOptionsForLingo` command for Lingo 0x0A, or does not return bit 00 set to 1, then it does not support the Digital Audio lingo `SetVideoDelay` command; go to Step 4.
3. If the Digital Audio lingo `SetVideoDelay` support bit (bit 00) is set to 1, the Apple device supports using the `SetVideoDelay` command to synchronize the video and digital audio streams. The accessory must use that command to maintain synchronization between the video and digital audio streams by sending a fixed time (in milliseconds) that the Apple device will add as a delay to its own video stream. The

accessory must also resend the command whenever its audio latency changes, for example at sample rate changes. The delay should be equal to the audio latency of the accessory; it corresponds to the difference between the time that digital audio data appears on the digital audio transport (USB) and the time the corresponding analog audio is sent to the speakers. There is no need to switch to an analog line out when video media is playing. After sending `SetVideoDelay`, go to Step 7.

Some Apple devices also support the accessory's setting the video stream delay by sending a `SetFIDTokenValues` command. This support is present if the `RetiPodOptionsForLingo` command returns a value of 1 in bit 03. Accessories should use the `SetFIDTokenValues` method if it is available, because it eliminates the need for sending `SetVideoDelay` unless the delay changes dynamically. See "Command 0x39: SetFIDTokenValues" (page 160) for details.

4. If the Apple device does not support the Digital Audio lingo `SetVideoDelay` command, the accessory must switch to its analog line out when video media is playing. The accessory must send a General lingo `GetiPodOptionsForLingo` command for the General lingo (0x00). If the Apple device returns a `RetiPodOptionsForLingo` command with bit 26 set to 1, continue to Step 5. If the Apple device does not support the `GetiPodOptionsForLingo` command for Lingo 0x00, or does not return bit 00 set to 1, then it does not support using `SetiPodPreferences` to switch between analog line out and USB Device mode audio routings; go to Step 6.

5. If the General lingo analog/USB Device mode Audio routing support bit (Bit 26) is set to 1, the Apple device supports using the `SetiPodPreferences` line out preference (class ID 0x03) to switch between analog line out and USB Device mode audio routings. The accessory does not need to identify or authenticate itself again. The accessory must include the digital audio lingo (0x0A) in its FID `IdentifyToken` and must set the analog/USB Device mode audio routing bit (bit 21) in its FID `AccessoryCapsToken`. Go to Step 7.

6. The Apple device does not support switching its audio routing using the `SetiPodPreferences` command. To switch between USB Device mode audio and analog line out audio routings, the accessory must identify and authenticate itself without declaring support for the Digital Audio lingo. Continue to Step 7.

7. Resume the IDPS process, including any FID token settings required in Step 5.

4.10.4.3 Line Level Output

Line level output and USB Device mode audio are mutually exclusive. All Apple device audio is sent to the LINE-OUT pins of the 30-pin connector until digital audio transport is enabled, at which point the Apple device's audio output is redirected to the USB Device mode audio streaming interface. When USB Device mode audio is disabled, Apple device audio is again sent to the LINE-OUT pins (see Table 4-1 (page 211)).

When disabling USB Device mode audio, the Apple device does not automatically enter the playback Pause state, but instead continues in its current Play or Pause state. The only difference is that the output mode changes to LINE-OUT. If appropriate, the accessory should set the Apple device to Pause when disabling digital audio.

4.10.4.4 Enabling and Disabling USB Device Mode Audio

USB Device mode audio is enabled and LINE-OUT audio is disabled when:

- Digital Audio lingo version 1.01: The accessory selects a nonzero-bandwidth alternate USB Device mode audio streaming interface and enters Extended Interface mode.

- Digital Audio lingo version 1.02 and later: The accessory replies to a `GetAccessorySampleRateCaps` command with a list of valid sample rates. Some Apple devices also support the accessory's providing a list of valid sample rates by sending a `SetFIDTokenValues` command. This support is present if the `RetiPodOptionsForLingo` command returns a value of 1 in bit 02. Accessories should use the `SetFIDTokenValues` method if it is available, because it eliminates the need for sending `RetAccessorySampleRateCaps` in response to `GetAccessorySampleRateCaps`. See "Command 0x39: SetFIDTokenValues" (page 160) for details.

USB Device mode audio is disabled and LINE-OUT audio is enabled when the accessory either reidentifies itself without supporting the Digital Audio lingo or disconnects from the USB Audio Configuration (for example, the user unplugs the USB cable).

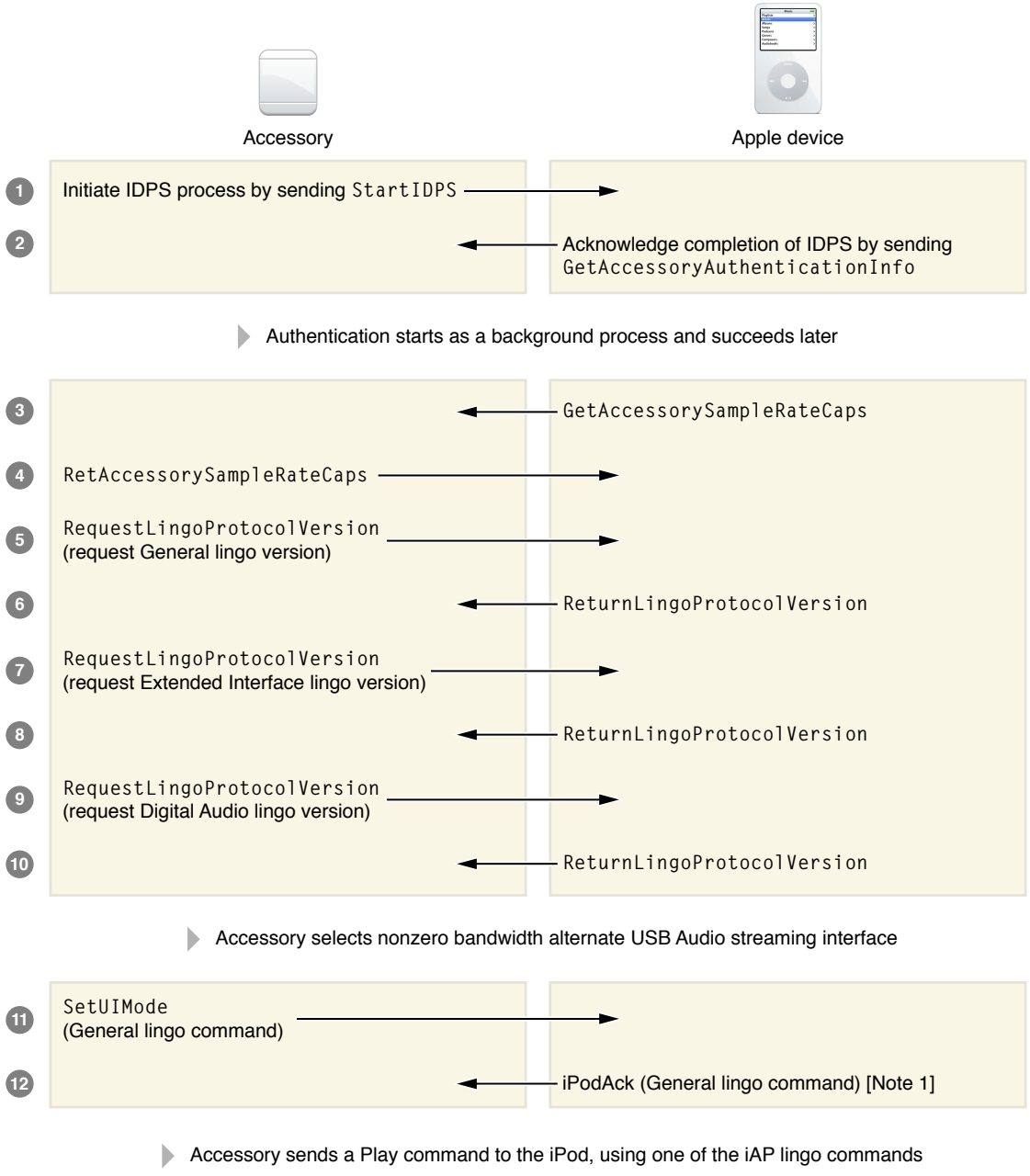
To enable USB Device mode audio, an Apple device and its attached accessory must perform the following steps, in the order listed, after the user has connected the Apple device to the accessory:

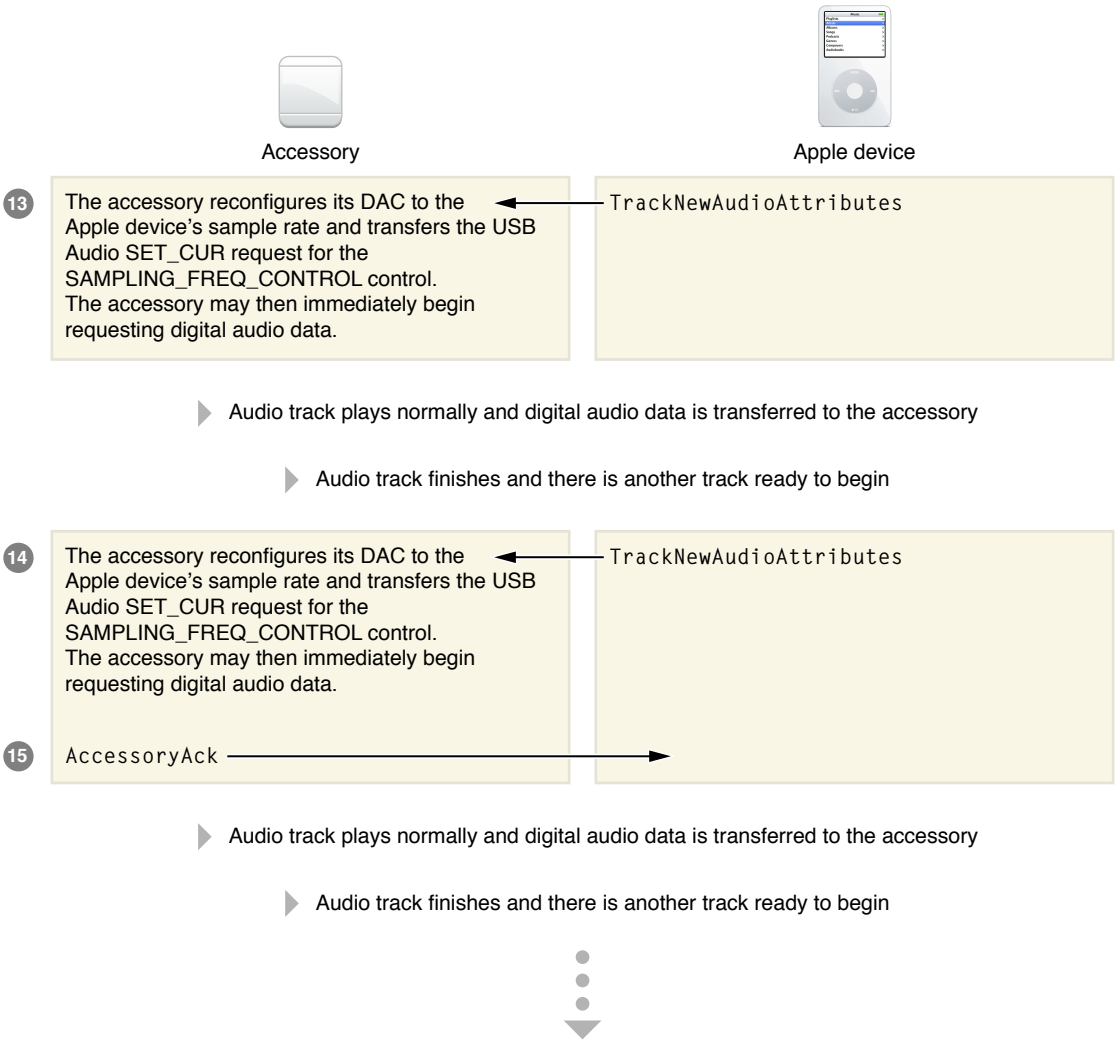
1. The Apple device's USB enumeration lists two configurations: 1 = Mass Storage, 2 = USB Audio and HID device.
2. The accessory sends the USB standard request `Set_Configuration` to select configuration 2.
3. The accessory completes the IDPS identification process, declaring the General lingo and the Digital Audio lingo, plus the Extended Interface lingo if the Apple device does not support Digital Audio lingo version 1.02. With Apple devices that do not support IDPS, the `IdentifyDeviceLingoes` command may be used. The Apple device authenticates the accessory for the requested lingoes.
4. The accessory selects a nonzero-bandwidth alternate USB audio streaming interface on the Apple device.
5. The accessory tells the Apple device to enter Extended Interface mode using the General lingo `SetUIMode` command (not required for Digital Audio lingo version 1.02). If an audio track is playing, playback will pause. If a video track is playing, playback will stop.
6. The accessory places the Apple device in the Play state.
7. The Apple device sends a `TrackNewAudioAttributes` command to the accessory.
8. The accessory acknowledges the `TrackNewAudioAttributes` command using the `AccessoryAck` command.
9. The accessory reconfigures its DAC to the Apple device's sample rate and transfers the USB Audio `SET_CUR` request for the `SAMPLING_FREQ_CONTROL` control, passing a sample rate equal to the value sent by the `TrackNewAudioAttributes` command.
10. The accessory requests digital audio packets from the isochronous USB pipe.
11. The audio track plays normally and digital audio data is transferred to the accessory.
12. The audio track finishes and there is another track ready to begin.
13. Steps 7 through 12 may be repeated indefinitely.

Note: Step 2 is slightly different if a 191 kΩ resistor is used as the identify resistor (see "Accessory Detect and Identify" in *MFi Accessory Hardware Specification*). In this case, the Apple device will present the USB audio and HID device configuration as the first configuration, and the accessory will be required to select configuration 1 through the `Set_Configuration` USB standard request.

An example of this process is diagrammed in Figure 4-1 (page 350), where the transactions shown assume that USB enumeration and accessory authentication are successful. An example of the Digital Audio identification process is shown in Table 4-295 (page 393).

Figure 4-1 Typical digital audio transactions between an Apple device and an accessory





Note 1: The `iPodAck` reply to a `SetUIMode` command may return a command pending acknowledgment along with a waiting time.

Note: With Digital Audio lingo version 1.01, switching from the zero bandwidth alternate USB audio streaming interface to the nonzero bandwidth interface without first exiting and entering Extended Interface mode can prevent a `TrackNewAudioAttributes` command from being sent from the Apple device when a track change occurs. The accessory must not select the zero-bandwidth alternate interface on the Apple device, even when changing sample rates, unless it is also exiting Extended Interface mode.

With Digital Audio lingo version 1.01, reenabling digital audio after an accessory has disabled it requires the following actions:

1. Enable the nonzero-bandwidth alternate USB audio streaming interface on the Apple device.
2. Enter Extended Interface mode using the General lingo `SetUIMode`. This will pause Apple device playback.
3. Place the Apple device in the Play state.

4. Receive a `TrackNewAudioAttributes` command from the Apple device.
5. Acknowledge the `TrackNewAudioAttributes` command using the `AccessoryAck` command.
6. Configure the accessory's playback DAC to the Apple device's sample rate.
7. Send a USB audio `SET_CUR` request for the `SAMPLING_FREQ_CONTROL` control, passing a sample rate equal to the value sent by the `TrackNewAudioAttributes` command.
8. Request digital audio packets from the isochronous USB pipe.

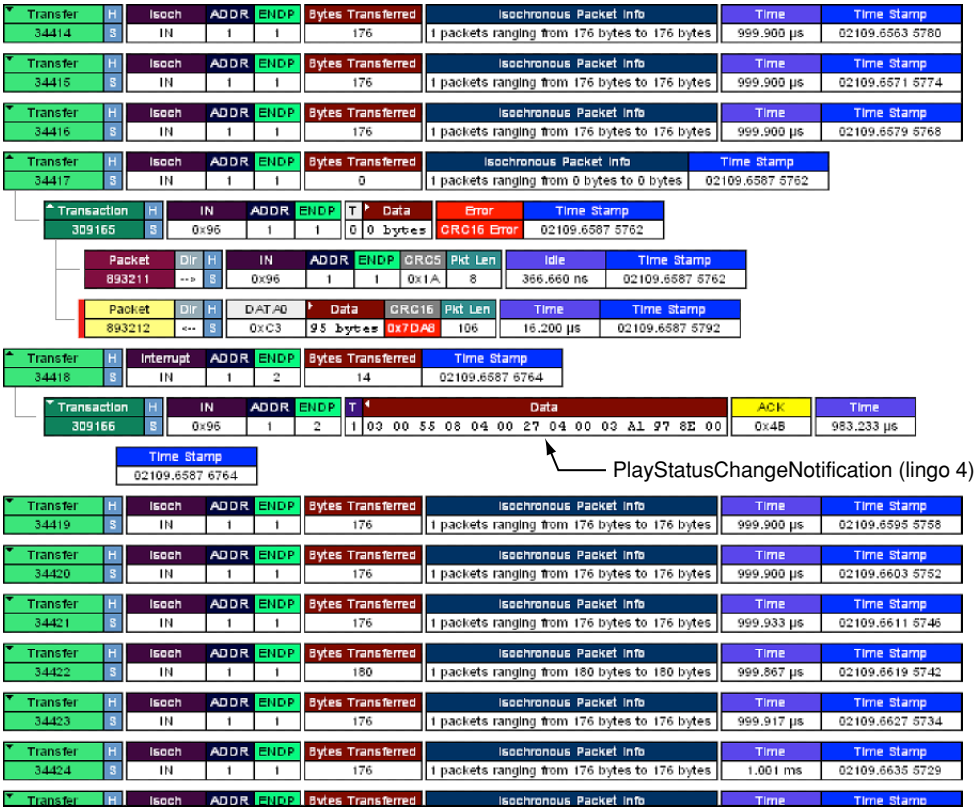
If the accessory requests digital audio data from the isochronous USB pipe before digital audio is enabled or before the correct digital sample rate has been negotiated, the Apple device will return isochronous packets filled with zeros. The Apple device will also return packets filled with zeros if authentication fails.

4.10.4.5 USB Device Mode Audio Errors on Older Apple Devices

On first-generation nano and 5G iPods, USB Device mode audio data from the Apple device will occasionally contain 16-bit cyclic redundancy check (CRC16) errors. If iAP commands are sent from the Apple device to a USB host while USB Device mode audio is enabled, there is a possibility that the USB audio DATA0 packet immediately preceding the iAP command will contain a CRC16 error. This error occurs infrequently; however, enabling Apple device notifications over USB (such as the Lingo 0x04 `PlayStatusChangeNotification` command, which is sent every 500 ms when enabled) greatly increases the possibility that a CRC16 error will occur.

To resolve this problem, the USB host must be able to recover immediately from a CRC16 error in the payload of the isochronous USB Device mode audio data. Figure 4-2 (page 353) illustrates a typical recovery sequence.

Figure 4-2 A USB host recovers from a CRC16 error



4.10.5 Command 0x00: AccessoryAck

Lingo: 0x0A — Origin: Accessory

The accessory sends the `AccessoryAck` command to acknowledge the receipt of a command from the Apple device and returns the command status (see Table 3-6 (page 125)). An `AccessoryAck` command should be sent only for commands whose documentation specifies that an `AccessoryAck` command is needed.

Table 4-234 `AccessoryAck` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command status; see Table 3-6 (page 125).
0x*NN*	1	The ID for the command being acknowledged
Packet footer specified in "Command Packets" (page 109).		

4.10.6 Command 0x01: iPodAck

Lingo: 0x0A — Origin: Apple device

The Apple device sends the `iPodAck` command when it receives an invalid or unsupported command or a bad parameter. The command status returns are the same as those used for the `AccessoryAck` command (see Table 3-6 (page 125)).

Table 4-235 `iPodAck` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command status; see Table 3-6 (page 125).
0x*NN*	1	The ID for the command being acknowledged
Packet footer specified in "Command Packets" (page 109).		

4.10.7 Command 0x02: GetAccessorySampleRateCaps

Lingo: 0x0A — Origin: Apple device

The Apple device uses this command to request the list of supported sample rates from the accessory. The Apple device sends it after the accessory indicates its support of the Digital Audio lingo during its identification process. The accessory responds to this command using the `RetAccessorySampleRateCaps` command.

After the Apple device sends a `GetAccessorySampleRateCaps` command, it waits up to 3 seconds for a `RetAccessorySampleRateCaps` command from the accessory before timing out. If a timeout occurs, the Apple device resends the command. It will resend the command up to three times if necessary, and if all three attempts fail, the Apple device will then respond to any digital audio request with zeros.

If the Apple device receives a `RetAccessorySampleRateCaps` command with invalid parameters, it sends the appropriate command status using an `iPodAck` command (see Table 3-6 (page 125)). At this point the accessory should send a `RetAccessorySampleRateCaps` with correct parameters. The Apple device allows up to three retries if necessary, and if all three attempts fail, the Apple device then responds to any digital audio request with zeros.

Table 4-236 `GetAccessorySampleRateCaps` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.10.8 Command 0x03: RetAccessorySampleRateCaps

Lingo: 0x0A — Origin: Accessory

An accessory sends the `RetAccessorySampleRateCaps` command in response to a `GetAccessorySampleRateCaps` command from an Apple device.

The accessory returns the list of sample rates it supports with this command. The sample rates must be taken from the list of Apple device sample rates shown in Table 4-238 (page 355). Any audio encoded at an unsupported sample rate is resampled internally by the Apple device to conform to a supported sample rate. In general, audio quality will be better when no resampling is performed; therefore accessories should support as many values listed in Table 4-238 (page 355) as possible.

Note: At a minimum, every accessory must support the sample rates 32 KHz, 44.1 KHz, and 48 KHz.

A `RetAccessorySampleRateCaps` command with sample rates not listed in Table 4-238 (page 355), or missing any of the required sample rates, is invalid. If the Apple device receives such a command, it sends the accessory an `iPodAck` command with a negative acknowledgment as the command status.

Table 4-237 `RetAccessorySampleRateCaps` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	*NN*	A list of *n* sample rates (32-bit big-endian format) supported by the accessory. The sample rates must be taken from Table 4-238 (page 355).
Packet footer specified in "Command Packets" (page 109).		

Table 4-238 Digital audio sample rates supported by Apple devices (in Hertz)

Decimal	Hexadecimal	Required/Optional
8000	0x00001F40	Optional
11025	0x00002B11	Optional
12000	0x00002EE0	Optional
16000	0x00003E80	Optional
22050	0x00005622	Optional
24000	0x00005DC0	Optional
32000	0x00007D00	Required
44100	0x0000AC44	Required
48000	0x0000BB80	Required

4.10.9 Command 0x04: TrackNewAudioAttributes

Lingo: 0x0A — Origin: Apple device

The Apple device sends the `TrackNewAudioAttributes` command before the first audio track begins playing. It sends the command again whenever it starts playing a track with different sample rate, sound check, or track volume parameters. In response to this command, accessories should prepare themselves to receive audio data with the new parameters.

The accessory must acknowledge this command, using the `AccessoryAck` command, but the Apple device does not wait for this acknowledgment before allowing digital audio to be transferred to the accessory.

The sample rate sent to the accessory is taken from the list of sample rates returned to the Apple device by the `RetAccessorySampleRateCaps` command. If the accessory supports the sample rate of the current audio track, then it is sent as the current sample rate. If the accessory does not support the sample rate, the Apple device resamples the audio data to a supported sample rate in real time and sends this new supported sample rate as the current sample rate.

The Sound Check value and track volume adjustment value are the corresponding values set by iTunes, rounded to the nearest integer. The values represent gain (in decibels) and may be positive or negative. Note that if the Sound Check option in the Apple device is disabled, the `TrackNewAudioAttributes` command always sends 0 as the new Sound Check value.

When the Apple device sends a `TrackNewAudioAttributes` command, it waits up to 500 ms for the `AccessoryAck` command before timing out. If a timeout occurs or if the `AccessoryAck` returns an error as the command status, the Apple device sends the command again.

IMPORTANT: The `TrackNewAudioAttributes` command should not be used as a general mechanism to detect track changes. Use appropriate commands from the Display Remote or Extended Interface lingoes instead.

Table 4-239 `TrackNewAudioAttributes` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	New sample rate.
0x*NN*	4	New Sound Check value.
0x*NN*	4	New track volume adjustment.
Packet footer specified in "Command Packets" (page 109).		

4.10.10 Command 0x05: SetVideoDelay

Lingo: 0x0A — Origin: Accessory

The accessory sends this command to inform the Apple device of a new delay (in milliseconds) that must be applied to Apple device video to synchronize it with the the audio for the currently playing video. See "Audio/Video Synchronization" (page 347). The Apple device responds to `SetVideoDelay` with a an "Command 0x01: iPodAck" (page 354) command.

IMPORTANT: Before sending this command, the accessory must verify that `RetiPodOptionsForLingo` reports that bit 00 for Lingo 0x0A is set; see Table 3-132 (page 192).

Table 4-240 `SetVideoDelay` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Video delay in milliseconds
Packet footer specified in "Command Packets" (page 109).		

4.11 Lingo 0x0C: Storage Lingo

The iAP Storage lingo, Lingo 0x0C, lets an accessory store files on an attached Apple device as if it were a hard drive. Use of the Storage lingo requires accessory authentication.

4.11.1 Command History of the Storage Lingo

Table 4-241 (page 357) shows the history of command changes in the Storage lingo:

Table 4-241 Storage lingo command history

Lingo version	Command changes	Features
1.01	Add: 0x00-0x02, 0x04, 0x07-08, 0x10-0x12	Support for iTunes Tagging.
1.02	Add:0x80-0x82. Add options to 0x12	Add support for Sports lingo.

4.11.2 Command Summary

Table 4-242 (page 357) lists the Storage lingo commands.

Table 4-242 Storage lingo commands

Cmd ID	Command name	Direction	Parameters:bytes
0x00	`iPodAck`	Dev to Acc	`{ackStatus:1, cmdID:1, handle:1, [transactionID:2]}`

Cmd ID	Command name	Direction	Parameters:bytes
0x01	GetiPodCaps	Acc to Dev	None
0x02	RetiPodCaps	Dev to Acc	{totalSpace:8, maxFileSize:4, maxWriteSize:2, Reserved:6, majorVersion:1, minorVersion:1}
0x03	Reserved		
0x04	RetiPodFileHandle	Dev to Acc	{handle:1}
0x05-0x06	Reserved		
0x07	WriteiPodFileData	Acc to Dev	{offset:4, handle:1, data:<var>}
0x08	CloseiPodFile	Acc to Dev	{handle:1}
0x09-0x0F	Reserved		
0x10	GetiPodFreeSpace	Acc to Dev	None
0x11	RetiPodFreeSpace	Dev to Acc	{freeSpace:8}
0x12	OpeniPodFeatureFile	Acc to Dev	{feature:0x*NN*}
0x13-0x7F	Reserved		
0x80	AccessoryAck	Acc to Dev	{ackStatus:1, cmdID:1, handle:1}
0x81	GetAccessoryCaps	Dev to Acc	None
0x82	RetAccessoryCaps	Acc to Dev	{Reserved:20, protocolVersion:2}
0x83-0xFF	Reserved		

The following is the typical sequence of commands used to store data on an Apple device:

1. Complete the accessory identification and authentication processes, as specified in "Transport Initialization" (page 88), making sure to identify the accessory as supporting the Storage lingo.
2. Send GetiPodCaps and receive RetiPodCaps, which returns the maximum write size.
3. Send OpeniPodFeatureFile and receive RetiPodFileHandle. The returned file handle points to a specific area in the Apple device's file system. Currently the only accessible area is /iPod_Control/Device/Accessories/Tags/, used by the Radio Tagging System (see "iTunes Tagging Accessory Design" (page 467)).
4. Send one or more WriteiPodFileData commands with the data to be stored. Receive an iPodAck command for each one.
5. Send CloseiPodFile when finished writing all the data. Alternately, all files are automatically closed when the accessory detaches.

Note: The accessory should use "Command 0x10: GetiPodFreeSpace" (page 362) to verify that there is adequate free space available on the Apple device's drive before attempting to open or write data to a file. If there is less than 5 MB available, then calls to `OpeniPodFeatureFile` or `WriteiPodFileData` may fail for lack of memory. The recommended practice is to check the free space before opening or writing data, and to handle any out-of-memory `iPodAck` commands by retaining the data in internal storage and displaying an "iPod Full" message on the accessory's display.

4.11.3 Command 0x00: iPodAck

Lingo: 0x0C — Origin: Apple device

The Apple device sends this command to acknowledge the receipt of a Storage lingo command from the accessory.

Table 4-243 Storage `iPodAck` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command status; see Table 3-6 (page 125).
0x*NN*	1	The ID for the command being acknowledged.
0x*NN*	1	The file handle for the command (0xFF if not applicable.)
Packet footer specified in "Command Packets" (page 109).		

4.11.4 Command 0x01: GetiPodCaps

Lingo: 0x0C — Origin: Accessory

The accessory asks the Apple device to return its storage capabilities. The Apple device replies with `RetiPodCaps`.

Table 4-244 `GetiPodCaps` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.11.5 Command 0x02: RetiPodCaps

Lingo: 0x0C — Origin: Apple device

The Apple device tells the accessory about the Apple device's storage capabilities:

- `totalSpace` is the amount of storage on the Apple device in bytes, including space currently in use.
- `maxFileSize` is the largest possible size, in bytes, of any file on the Apple device.
- `maxWriteSize` is the largest amount of data, in bytes, that can be written to the Apple device in a single `WriteiPodFileData` command.
- `majorVersion` and `minorVersion` are the version number of the Storage lingo protocol implemented by the accessory.

Table 4-245 `RetiPodCaps` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	8	`totalSpace`. The total amount of storage space on the Apple device in bytes, including space currently in use.
0x*NN*	4	`maxFileSize`. The size in bytes of the largest possible file on the Apple device.
0x*NN*	2	`maxWriteSize`. The size in bytes of the largest possible amount of data than can be sent to the Apple device with `WriteiPodFile`.
0x*NN*	6	Reserved (6 bytes).
0x*NN*	1	`majorVersion` (1 byte). The major version number.
0x*NN*	1	`minorVersion` (1 byte). The minor version number.
Packet footer specified in "Command Packets" (page 109).		

4.11.6 Command 0x04: RetiPodFileHandle

Lingo: 0x0C — Origin: Apple device

The Apple device returns a unique 8-bit handle to identify the file.

The value of `handle` is a session-specific identifier that represents a file. This is similar to a Unix file descriptor. The handle is valid until either the accessory is detached or the accessory sends a `CloseiPodFile` command with this handle.

Table 4-246 `RetiPodFileHandle` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`handle` (1 byte)
Packet footer specified in "Command Packets" (page 109).		

4.11.7 Command 0x07: WriteiPodFileData

Lingo: 0x0C — Origin: Accessory

The accessory writes a block of data to a file starting at the offset passed with this command. The file must have been previously opened for writing; failure is reported as a bad parameter (see Table 3-6 (page 125)). If the file was not previously opened for writing, or the handle is invalid (invalid handle error), or the `writeSize` exceeds the Apple device's capabilities (bad parameter error), then the operation will fail. If the caller attempts to write too much data, the state of the file will be undefined; some or none of the data may have been added to the file.

All data must be written sequentially, but write actions may occur across multiple `WriteiPodFileData` commands. The amount of data transferred must also be within the bounds set by the Apple device's capabilities. If any of these criteria are not met, the Apple device will return an `iPodAck` command with a failure message.

When a `WriteiPodFileData` command fails, the state of the file is left unknown. The accessory must close the file and then create a new file to write the data.

`WriteiPodFileData` passes the following parameters:

- `offset` is the offset (in bytes) into the file at which to begin writing.
- `handle` is a unique file identifier.
- `data` is the data to be written to the file.

Table 4-247 `WriteiPodFileData` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	`offset`. The offset into the file at which to begin writing, in bytes.
0x*NN*	1	`handle` (1 byte).
0x*NN*	*NN*	`data` (variable length). The file data.
Packet footer specified in "Command Packets" (page 109).		

4.11.8 Command 0x08: CloseiPodFile

Lingo: 0x0C — Origin: Accessory

This command closes a file and releases its handle. The handle is invalid for further use after this call, and the Apple device may assign the handle later to represent a different file. an `iPodAck` command is sent on success or failure. Parameter `handle` is a unique file identifier.

Table 4-248 `CloseiPodFile` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`handle` (1 byte).
Packet footer specified in "Command Packets" (page 109).		

4.11.9 Command 0x10: GetiPodFreeSpace

Lingo: 0x0C — Origin: Accessory

The accessory asks the Apple device to return the amount of free space on its storage system. The Apple device replies with `RetiPodFreeSpace`.

Table 4-249 `GetiPodFreeSpace` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.11.10 Command 0x11: RetiPodFreeSpace

Lingo: 0x0C — Origin: Apple device

The Apple device tells the accessory the current amount of free space (in bytes) in its storage system.

Table 4-250 `RetiPodFreeSpace` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	8	`freeSpace`. The amount of the Apple device's free space.
Packet footer specified in "Command Packets" (page 109).		

4.11.11 Command 0x12: OpeniPodFeatureFile

Lingo: 0x0C — Origin: Accessory

The accessory uses the `OpeniPodFeatureFile` command to open a feature file on the Apple device. Table 4-251 (page 363) presents the format of the `OpeniPodFeatureFile` command packet.

In response, the Apple device returns a `RetiPodFileHandle` command with the feature file handle. If the feature type is not supported or the feature parameters are not valid, an `iPodAck` with bad parameter error status is returned instead of `RetiPodFileHandle`.

IMPORTANT: Feature files of different types can be open at the same time, but only one file of each type can be open. To open a second file of a given type, the first file must be closed.

Table 4-251 `OpeniPodFeatureFile` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	featureType
0x*NN*	4	fileOptionsMask
0x*NN*	*NN*	`fileData` (maximum 128 bytes) data to be appended at close. `fileOptionsMask` bit 0 must be set if this field is nonzero length.
Packet footer specified in "Command Packets" (page 109).		

Table 4-252 (page 363) lists the current possible `featureType` values for the `OpeniPodFeatureFile` command packet.

Table 4-252 featureType values

featureType	Description
0x00	Reserved
0x01	Radio tagging (no `fileOptionsMask` or `fileData` can be passed)
0x02	Cardio equipment workout; requires options to be set as specified in "Recording Workout Data" (page 505).
0x03-0xFF	Reserved

Table 4-253 (page 363) lists the current possible `fileOptionsMask` values for the `OpeniPodFeatureFile` command.

Table 4-253 fileOptionsMask values

fileOptionsMask	Description
bit 0	1 = On feature file close or accessory detach, append the `fileData` binary file data bytes to the file. Note that `fileData` must be nonzero length if this bit is set. 0 = Do not append any bytes to the file. `fileData` must be zero length if this bit is zero.

fileOptionsMask	Description
bit 1	1 = On file close or accessory detach, append the XML <ipodInfo> element to the file (includes <openTime>, <closeTime>, <model>, <softwareVersion>, and <serialNumber>). This option is applied before the bit 0 option. 0 = Do not write <ipodInfo> element to file. This option must be set when the XML Signature (bit 3) option is set = 1.
bit 2	Reserved (must be 0).
bit 3	1 = On file close or accessory detach, insert an XML <Signature> element to the file. This option is applied after bit 0 and bit 1 options. 0 = Do not insert an XML <Signature> element. When this option bit is set, the <ipodInfo> option (bit 1) must be set = 1.
bits 31:4	Reserved (must be 0).

4.11.12 Command 0x80: AccessoryAck

Lingo: 0x0C — Origin: Accessory

The accessory must send the `AccessoryAck` command to acknowledge the receipt of a Storage lingo command from the Apple device. The accessory must respond with bad parameter status whenever a Storage lingo command in the range 0x83 to 0xFF is received from the Apple device.

Table 4-254 (page 364) shows the format of the `AccessoryAck` command packet.

Table 4-254 `AccessoryAck` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command status; see Table 3-6 (page 125).
0x*NN*	1	ID of Command being acknowledged
0xFF	1	File Handle for Command (set to 0xFF for not applicable)
Packet footer specified in "Command Packets" (page 109).		

4.11.13 Command 0x81: GetAccessoryCaps

Lingo: 0x0C — Origin: Apple device

The `GetAccessoryCaps` command may be sent by the Apple device to ask the accessory to return its storage capabilities (if any). Table 4-255 (page 365) presents the format of the `GetAccessoryCaps` command packet. The accessory replies with `RetAccessoryCaps`.

Table 4-255 `GetAccessoryCaps` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

4.11.14 Command 0x82: RetAccessoryCaps

Lingo: 0x0C — Origin: Accessory

The accessory must send the `RetAccessoryCaps` command in response to a `GetAccessoryCaps` command from the Apple device to determine the storage capabilities supported by the accessory. Table 4-256 (page 365) presents the format of the `RetAccessoryCaps` command packet.

Table 4-256 `RetAccessoryCaps` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x00	1	Reserved (must be 0x00)
0xFF	1	Reserved (must be 0xFF)
0x*NN*	1	Major Storage lingo protocol version supported by the accessory (currently 0x01)
0x*NN*	1	Minor Storage lingo protocol version supported by the accessory (currently 0x02)
Packet footer specified in "Command Packets" (page 109).		

4.12 Lingo 0x0D: iPod Out Lingo

The iAP iPod Out lingo supports iPod Out mode, an Apple device operating mode described in "iPod Out Mode" (page 78).

4.12.1 Command History of the iPod Out Lingo

Table 4-257 (page 365) shows the history of changes to the iPod Out lingo.

Table 4-257 iPod Out lingo command history

Lingo version	Command changes	Features
1.00	Add: 0x00–0x05 and 0x0F	iPod Out support

4.12.2 iPod Out Lingo Commands

An accessory can use the iPod Out lingo commands to set up and manage iPod Out mode before or after the Apple device enters the mode. Table 4-258 (page 366) summarizes the iPod Out lingo commands.

Table 4-258 iPod Out lingo command summary

Command name	Cmd ID	Direction	Parameters:bytes
`iPodAck`	0x00	Dev to Acc	`{ackStatus:1, origCmdID:1}`
`GetiPodOutOptions`	0x01	Acc to Dev	`{enabledOptions:1}`
`RetiPodOutOptions`	0x02	Dev to Acc	`{enabledOptions:1, optionsBits:4}`
`SetiPodOutOptions`	0x03	Acc to Dev	`{optionsBits:4}`
`AccessoryStateChangeEvent`	0x04	Acc to Dev	`{stateChange:1}`

4.12.3 Command 0x00: iPodAck

Lingo: 0x0D — Origin: Apple device

The Apple device sends this command to acknowledge the receipt of an iPod Out command from the attached accessory and report the status of its execution.

Table 4-259 `iPodAck` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command status; see Table 3-6 (page 125).
0x*NN*	1	`origCmdID`: ID of the iPod Out command being acknowledged
Packet footer specified in "Command Packets" (page 109).		

4.12.4 Command 0x01: GetiPodOutOptions

Lingo: 0x0D — Origin: Accessory

The accessory sends this command to query the capabilities of the Apple device while it is in iPod Out mode. The Apple device responds with a `RetiPodOutOptions` command. If the accessory passes 0x00 in the `enabledOptions` parameter, the Apple device reports all possible iPod Out options. If it passes 0x01 in `enabledOptions`, the Apple device reports only its currently set iPod Out options.

Table 4-260 `GetiPodOutOptions` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`enabledOptions`: 0x00 = report all iPod Out options the Apple device is capable of supporting; 0x01 = report only the iPod Out options currently set on the Apple device; 0x02-0xFF = reserved.
Packet footer specified in "Command Packets" (page 109).		

4.12.5 Command 0x02: RetiPodOutOptions

Lingo: 0x0D — Origin: Apple device

The Apple device sends this command to the accessory to report its iPod Out output capabilities, by setting bits in the big-endian `optionsBits` parameter. It also copies back the value of the `enabledOptions` parameter that the accessory sent in its `GetiPodOutOptions` command. If the value of `enabledOptions` is 0x01, only options that are currently set are reported.

Table 4-261 `RetiPodOutOptions` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`enabledOptions`: 0x00 = reporting all iPod Out options the Apple device can support; 0x01 = reporting only the iPod Out options currently set on the Apple device; 0x02-0xFF = reserved.
0x*NN*	4	`optionsBits`: Bits set to 1 to report iPod Out output options; all other bits are set to 0. See Table 4-262 (page 367).
Packet footer specified in "Command Packets" (page 109).		

Table 4-262 Options for the iPod Out output from the Apple device

Bit	Meaning
00	Display audio content, including all media that don't have moving video, including songs, audio podcasts, audio books, and music videos played as songs only. `SetiPodOutOptions` cannot disable this option.
01	Display incoming phone call user interface.
02	Display SMS/MMS incoming text interface.
03	Reserved
04	Display incoming voicemail user interface.

Bit	Meaning
05	Display incoming push notification user interface.
06	Display clock alarm notification user interface.
07	Reserved
08	Enable test pattern iPodOut UI to cause the iPodOut video output to display test patterns as an aid in iPodOut development. The patterns can be cycled via the Simple Remote iPodOut buttons and wheel. When used, this option must be set before iPodOut is entered via the General Lingo `SetUIMode` command.) Important: The accessory must use all the test patterns generated by this option to verify that the entire video output is visible. This option must not be enabled in a shipping product.
09	Show minimal iPodOut UI; the iPodOut UI displayed over TV Out shows the minimum amount of UI to remain functional. If set, this overrides bit 10. If neither bit 09 or bit 10 is set, the recommended iPodOut UI is displayed.
10	Show full iPodOut UI; the iPodOut UI displayed over TV Out shows all available UI features. This option is overridden if bit 09 is also set. In iOS 5, recommended UI (bit 09 and 10 both not set) and full UI are equivalent.
11:31	Reserved

4.12.6 Command 0x03: SetiPodOutOptions

Lingo: 0x0D — Origin: Accessory

The accessory sends this command to the iPod to limit the Apple device's iPod Out options. This lets the accessory control what content the user can ask it to display, in case there are legal regulations that the accessory needs to follow (for example, if it has a video screen visible to the driver of a car). The Apple device replies with an `iPodAck` command.

Table 4-263 `SetiPodOutOptions` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`optionsBits` (big-endian): Bits set to 1 to enable iPod Out output options; all other bits set to 0 to disable options. See Table 4-262 (page 367).
0x*NN*	3	`optionsBits`
Packet footer specified in "Command Packets" (page 109).		

4.12.7 Command 0x04: AccessoryStateChangeEvent

Lingo: 0x0D — Origin: Accessory

The accessory sends this command to tell the Apple device that the accessory's state is changing. The Apple device responds by sending an `iPodAck` command, regardless of whether or not the Apple device acts on this information.

Table 4-264 `AccessoryStateChangeEvent` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`stateChange`: Accessory state transition; see Table 4-265 (page 369).
Packet footer specified in "Command Packets" (page 109).		

Table 4-265 Accessory state transitions

Value	Meaning
0x00	Accessory display switching away from iPod Out
0x01	Accessory display switching back to iPod Out
0x02	Accessory audio switching away from iPod Out
0x03	Accessory audio switching back to iPod Out
0x04	Accessory entering "daytime" mode (full display brightness)
0x05	Accessory entering "nighttime" mode (display dimmed)
0x06-0xFF	Reserved

4.13 Lingo 0x0E: Location Lingo

The National Marine Electronics Association (NMEA) has established a standard interface for transferring global positioning system (GPS) information. The NMEA *0183 Interface Standard,* Version 3.01 (released 2002/01), gives complete details about the sentence formats and contents.

The iAP Location lingo provides a mechanism by which NMEA sentences and other types of location information can be sent from accessories to attached Apple devices. Thus the Location lingo lets an accessory that generates raw location data send the data to an Apple device that collects, interprets, and displays it. Currently, only iOS 3.0 and later versions (including iOS 4) support the Location lingo.

Note: If an accessory provides or receives any kind of location data to or from an Apple device, it must use the Location lingo in addition to any other lingoes it supports.

4.13.1 Location Data Requirements

Accessories that generate NMEA GPS location sentences must support at least the GPGGA sentence type, as defined in the *0183 Interface Standard*. Apple devices also accept the GPRMC sentence type; if the accessory supports it, the accessory must provide it to the Apple device when using the Location lingo. The accessory may support additional NMEA sentence types, either standard or proprietary, and future Apple device models may also support more sentence types.

For maximum quality of the user experience, accessories that generate NMEA GPS location sentences must enable filtering and pass to the Apple device only sentence types that are in the filter list. Whenever possible, Apple devices try to conserve power by using location data from an external accessory instead of using on-board positioning technologies.

4.13.2 Power from the Apple Device for Accessories Using the Location Lingo

Attached accessories that support the Location lingo are notified of Apple device state changes, such as transitions between Power On and Hibernate. The accessory must keep its power consumption below the maximum allowed limits for each Apple device state; see "Apple Device Power States" in *MFi Accessory Hardware Specification*. Current on the Accessory Power line of the 30-pin connector will be off when the Apple device hibernates; when the Apple device wakes, the accessory must identify and authenticate itself, using the IDPS process specified in "Transport Initialization" (page 88).

Accessories that support the Location lingo may enter Intermittent High power mode as specified in the section "Accessory Power Policy" in *MFi Accessory Hardware Specification*. They may then draw high power after receiving a `SetAccessoryControl` command specifying GPS radio power on (see Table 4-279 (page 381)). Accessories must reduce their power consumption to low power mode within 1 second of receiving a `SetAccessoryControl` command specifying GPS radio power off. Regardless of the state of power set by the Location lingo, accessories must comply with Apple device state changes as detailed in "Apple Device Power States" in *MFi Accessory Hardware Specification*.

4.13.3 Command Summary

Certain prerequisites must be met for an accessory to use the Location lingo:

- The accessory must identify itself through the Identify Device Preferences and Settings (IDPS) process, as specified in "Identification" (page 93).
- To send and receive Location lingo and IDPS commands, the accessory must be able to generate and interpret transaction IDs, as described in "Transaction IDs" (page 110), throughout the communication session.
- The accessory must perform Authentication 2.0, as described in "Authentication" (page 103). Location lingo commands are usable after the accessory enters the background authentication state.

Note: With Apple devices that support wireless iAP over Bluetooth, an accessory need not be physically attached to an Apple device to send it location information.

The Apple device determines accessory capabilities by sending the Location lingo `GetAccessoryCaps` command. A type parameter in this command specifies the type of capabilities queried. Capabilities type 0x00 requests the types of location services that the accessory supports and types 0x01–0x3F request details about the specific location services. Location capabilities include the following:

- Accessory general capabilities
- NMEA GPS location information (sentence filtering and GPGGA sentence generation are required).
- Location assistance information
- Asynchronous location notification control
- Capabilities specific to each location information type

Location lingo commands are extensible; new features, capabilities, and parameters may be added to future lingo versions. Lingo extensions will be made backward compatible so that existing implementations will continue to work. Accessory implementations must comply with the following rules to be compatible with future lingo versions:

- For all command responses, check for a packet length greater than or equal to the expected length. Do not check for an exact payload length, because this will fail after extensions are made. If the packet length is greater than expected, ignore any additional data.
- When reading lingo bitfield parameters, clear unrecognized bits to 0 before testing for feature bits.
- Latitude location data from an accessory must be within the range –90 to +90 degrees.
- Longitude location data from an accessory must be within the range –180 to +180 degrees.
- For other parameters, do range checks on their values and return an `AccessoryAck` command with a bad parameter status (0x04) if a parameter is out of range.

Note: Apple devices and accessories that support the Location lingo must initialize their control states to empty or disabled when the device-accessory connection is first established. Initializing the control state is effectively the same as receiving `SetAccessoryControl` commands for all `locType` values, with `ctlData` set to 0x0 for each. This will ensure that for all `locType` values, notifications are disabled by default and accessories are in low power mode, as defined in the section "Accessory Power Policy" in *MFi Accessory Hardware Specification*. One or more control states may be changed later, depending on the Apple device's or accessory's capabilities and usage requirements.

Table 4-266 (page 371) lists the Location lingo commands.

Table 4-266 Location lingo commands

Cmd ID	Command name	Direction	Parameters:bytes
0x00	`AccessoryAck`	Acc to Dev	`{transID:2, cmdStatus:1, cmdIDOrig:1}`
0x01	`GetAccessoryCaps`	Dev to Acc	`{transID:2, locType:1}`

Cmd ID	Command name	Direction	Parameters:bytes
0x02	RetAccessoryCaps	Acc to Dev	{transID:2, locType:1, capsData:<var>}
0x03	GetAccessoryControl	Dev to Acc	{transID:2, locType:1}
0x04	RetAccessoryControl	Acc to Dev	{transID:2, locType:1, ctlData:<var>}
0x05	SetAccessoryControl	Dev to Acc	{transID:2, locType:1, ctlData:<var>}
0x06	GetAccessoryData	Dev to Acc	{transID:2, locType:1, dataType:1}
0x07	RetAccessoryData	Acc to Dev	{transID:2, locType:1, dataType:1, sectCur:2, sectMax:2, locData:<var>}
0x08	SetAccessoryData	Dev to Acc	{transID:2, locType:1, dataType:1, sectCur:2, sectMax:2, locData:<var>}
0x09	AsyncAccessoryData	Acc to Dev	{transID:2, locType:1, dataType:1, sectCur:2, sectMax:2, locData:<var>}
0x0A–0x7F	Reserved		
0x80	iPodAck	Dev to Acc	{transID:2, cmdStatus:1, cmdIDOrig:1}
0x81–0xFF	Reserved		

4.13.4 A Typical Location Data Session

Table 4-267 (page 372) shows a typical exchange of commands by which an Apple device may obtain location data from an accessory. The first time such session could take place would be after the Apple device had acknowledged successful completion of the accessory identification process by sending IDPSStatus (see "The IDPS Process" (page 94)) and had begun the process of authenticating the accessory, as specified in "Authentication" (page 103).

Table 4-267 Sample command interchange

Step	Command	locType	dataType	Direction	Comments
1	GetAccessoryCaps	0x00		Dev to Acc	The Apple device queries the accessory's system capabilities.
2	RetAccessoryCaps	0x00		Acc to Dev	The accessory returns its system capabilities.
3	GetAccessoryCaps	0x01		Dev to Acc	The Apple device queries the accessory's NMEA GPS location capabilities.
4	RetAccessoryCaps	0x01		Acc to Dev	The accessory returns its NMEA GPS location capabilities.

Step	Command	locType	dataType	Direction	Comments
5	GetAccessoryCaps	0x02		Dev to Acc	The Apple device queries the accessory's location assistance capabilities.
6	RetAccessoryCaps	0x02		Acc to Dev	The accessory returns its location assistance capabilities.
7	GetAccessoryControl	0x00		Dev to Acc	The Apple device queries the accessory's system control state.
8	RetAccessoryControl	0x00		Acc to Dev	The accessory returns its system control state.
9	GetAccessoryControl	0x01		Dev to Acc	The Apple device queries the accessory's NMEA GPS location control state.
10	RetAccessoryControl	0x01		Acc to Dev	The accessory returns its NMEA GPS location control state.
11	SetAccessoryControl	0x00		Dev to Acc	The Apple device sets the accessory's system control state for locType = 0x00 in accordance with the capabilities and controls sent to it by previous RetAccessoryCaps and RetAccessoryControl commands.
12	AccessoryAck			Acc to Dev	The accessory acknowledges receipt of SetAccessoryControl.
13	SetAccessoryData	0x01	0x00	Dev to Acc	The Apple device sets the NMEA sentence filter list string.
14	AccessoryAck			Acc to Dev	The accessory acknowledges receipt of SetAccessoryData.
15	SetAccessoryControl	0x01		Dev to Acc	The Apple device sets the accessory's system control state for locType = 0x01 in accordance with the capabilities and controls sent to it by previous RetAccessoryCaps and RetAccessoryControl commands.
16	AccessoryAck			Acc to Dev	The accessory acknowledges receipt of SetAccessoryControl.

Step	Command	locType	dataType	Direction	Comments
17	GetAccessoryData	0x02	0x03	Dev to Acc	The Apple device requests the satellite ephemeris maximum refresh interval.
18	RetAccessoryData	0x02	0x03	Acc to Dev	The accessory returns the satellite ephemeris maximum refresh interval.
19	GetAccessoryData	0x02	0x04	Dev to Acc	The Apple device requests the satellite ephemeris recommended refresh interval.
20	RetAccessoryData	0x02	0x04	Acc to Dev	The accessory returns the satellite ephemeris recommended refresh interval.
21	AsyncAccessoryData	0x02	0x05	Acc to Dev	The accessory sends a request for current GPS time of the system.
22	iPodAck			Dev to Acc	The Apple device acknowledges receipt of the GPS time request.
23	AsyncAccessoryData	0x02	0x02	Acc to Dev	The accessory sends an ephemeris URL to request an update to its ephemeris data.
24	iPodAck			Dev to Acc	The Apple device acknowledges receipt of the satellite ephemeris data URL string.
25	SetAccessoryData	0x02	0x05	Dev to Acc	The Apple device sets the current GPS time of the accessory.
26	AccessoryAck			Acc to Dev	The accessory acknowledges receipt of SetAccessoryData.
27	SetAccessoryData	0x02	0x00	Dev to Acc	The Apple device sets the current location data on the accessory for use in looking up the satellite position within the ephemeris.
28	AccessoryAck			Acc to Dev	The accessory acknowledges receipt of SetAccessoryData.
29	SetAccessoryData	0x02	0x01	Dev to Acc	The Apple device sets the accessory's ephemeris with data downloaded from the URL provided in Step 23.
30	AccessoryAck			Acc to Dev	The accessory acknowledges receipt of SetAccessoryData.
If multisection data, repeat Steps 29–30 as needed. See "Multisection Data Transfers" (page 116).					

Step	Command	locType	dataType	Direction	Comments
31	AsyncAccessoryData	0x01	0x80	Acc to Dev	The accessory begins sending location data to the Apple device.
32	iPodAck			Dev to Acc	The Apple device acknowledges receipt of AsyncAccessoryData.
If multisection data, repeat Steps 31–32 as needed. See "Multisection Data Transfers" (page 116).					

4.13.5 Command 0x00: AccessoryAck

Lingo: 0x0E — Origin: Accessory

This command is sent by the accessory in response to a command sent from the Apple device. It has two forms:

- The Section form is sent if the command from the Apple device is part of a multisection data transfer, as described in "Multisection Data Transfers" (page 116). This form of the command is discussed in "Multisection AccessoryAck Packet" (page 375).
- The Default form, shown in Table 4-268 (page 375), is sent under these conditions:
  - When a bad parameter is received, an unsupported or invalid command is received, or a command that does not return any data has completed.
  - To indicate completion of a multisection data transfer, as described in "Multisection Data Transfers" (page 116).

Table 4-268 Default AccessoryAck packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command status; see Table 3-6 (page 125).
0x*NN*	1	cmdIDOrig: Original command ID for which this response is being sent.
Packet footer specified in "Command Packets" (page 109).		

4.13.5.1 Multisection AccessoryAck Packet

Lingo: 0x0E — Origin: Accessory

The accessory must send an AccessoryAck command in response to each multisection data transfer packet sent by the Apple device via SetAccessoryData. The accessory may send either of two forms of the AccessoryAck packet:

- When it has successfully received any section before the last section, the accessory must send the packet shown in Table 4-269 (page 376). The Apple device will send a new packet immediately.

- When it has successfully received the last section, the accessory must send the packet shown in Table 4-268 (page 375) with `ackStatus` = 0x00, indicating that the multisection data transfer is complete. The accessory must also send an acknowledgment of this form, with a nonzero `ackStatus`, whenever it has detected an error in receiving any packet.

Table 4-269 Multisection AccessoryAck packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x13	1	`ackStatus`: Section received successfully
0x08	1	`cmdIDOrig`: ID of command (`SetAccessoryData`) for which this response is being sent.
0x*NN*	2	`sectCur`: Section index for which this response is being sent.
Packet footer specified in "Command Packets" (page 109).		

4.13.6 Command 0x01: GetAccessoryCaps

Lingo: 0x0E — Origin: Apple device

This command is sent by the Apple device to get the accessory's capabilities. In response, the accessory sends a `RetAccessoryCaps` command with the same `transactionID`, the capability type, and the requested capability data. If a specified `locType` is not supported, an `AccessoryAck` must be returned with the bad parameter (0x04) status. Accessory support for capability type 0x00 (system caps) is mandatory; other capability types are optional. The Apple device requests the system caps from the accessory to determine its lingo version, system capabilities, and the other location types supported by the accessory. The accessory must send a corresponding `RetAccessoryCaps` within 500 ms in response to the `GetAccessoryCaps` command from the Apple device.

Table 4-270 GetAccessoryCaps packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`locType`: The capability type; see Table 4-271 (page 376).
Packet footer specified in "Command Packets" (page 109).		

Table 4-271 Values for locType

Value	Meaning
0x00	System capabilities; see Table 4-273 (page 377).
0x01	NMEA GPS location capabilities; see Table 4-274 (page 378).

Value	Meaning
0x02	Location assistance capabilities; see Table 4-275 (page 378).
0x03–0x3F	Reserved
0x40–0xFF	Invalid

4.13.7 Command 0x02: RetAccessoryCaps

Lingo: 0x0E — Origin: Accessory

The accessory sends this command in response to a `GetAccessoryCaps` command from the Apple device, returning the transaction ID and the requested capabilities type. It informs the Apple device of its capabilities in `capsData`. If an accessory does not support a particular capability type, it must return an `AccessoryAck` command with a bad parameter (0x04) status.

The `capsData` fields may be extended in the future as new features are added to the Location lingo. Apple devices receiving this command check for a minimum packet payload length (rather than an exact payload length) and parse only the supported fields, ignoring any additional data that may be appended.

Table 4-272 RetAccessoryCaps packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`locType`: The capability type; see Table 4-271 (page 376).
0x*NN*	*NN*	`capsData`: Capabilities data; see Table 4-273 (page 377), Table 4-274 (page 378), and Table 4-275 (page 378). The contents and length of this parameter depend on the value of `locType`.
Packet footer specified in "Command Packets" (page 109).		

Table 4-273 System capabilities values (locType = 0x00)

Name	Size in bytes	Bits	Meaning
`devVerMajor`	1	Bits 7:0	Location lingo major version supported by the accessory
`devVerMinor`	1	Bits 7:0	Location lingo minor version supported by the accessory
`sysCapsMask`	8	System capabilities (big-endian):	
		Bit 00	1 = Power management control support; Apple device-powered accessories must set this bit to allow power control by the Apple device.
		Bit 01	Reserved; set to 0.

Name	Size in bytes	Bits	Meaning
		Bit 02	1 = Asynchronous location notification support; the accessory generates asynchronous location data and does not need to be polled. This bit must be set to 1.
		Bits 63:03	Reserved; set to 0.
`locCapsMask`	8	Location type support (big-endian):	
		Bit 00	1 = System support; must be set to 1.
		Bit 01	1 = NMEA GPS location support (see **Note** below)
		Bit 02	1 = Location assistance support; either this bit or bit 01 must be set
		Bits 63:03	Reserved; set to 0.

Note: Accessories that generate NMEA GPS location sentences must support at least the GPGGA sentence type. The Apple device also accepts the GPRMC sentence type; if the accessory supports it, the accessory must provide it to the Apple device when using the Location lingo. The accessory may support additional NMEA sentence types, either standard or proprietary. Sentences must be generated at intervals compliant with the NMEA *0183 Interface Standard,* unless overridden by system controls or NMEA sentence filtering.

Table 4-274 NMEA GPS location capabilities values (locType = 0x01)

Name	Size in bytes	Bits	Meaning
`nmeaGpsCaps`	8	Bit 00 (big-endian)	Must be set to 1 to indicate NMEA GPS sentence filtering supported; the accessory must support a minimum filter list string length of 128 bytes.
		Bits 63:01	Reserved; set to 0.

Table 4-275 Location assistance capabilities values (locType = 0x02)

Name	Size in bytes	Bits	Meaning
`locAsstData`	8	Location assistance data types supported; the accessory's most recent `RetAccessoryCaps` command must set the `sysCapsMask` capability bit 01 if one or more of the following bits are set (see Table 4-273 (page 377)):	
		Bit 00 (big-endian)	Reserved; set to 0.
		Bit 01	Satellite ephemeris data required, for faster location fix. This bit must be set if bit 02 is set.

Name	Size in bytes	Bits	Meaning
		Bit 02	Satellite ephemeris data URL string (RFC 1738 compliant). This bit must be set if bit 01 is set. The accessory must supply a Web URL string for the Apple device to use to download ephemeris data and must accept that data when it is set by the Apple device (see "Command 0x06: GetAccessoryData" (page 382) and "Command 0x08: SetAccessoryData" (page 384)).
		Bit 03	Satellite position ephemeris data maximum refresh interval. If the accessory's long-term ephemeris degrades over time, this indicates the maximum amount of time during which downloaded data will be valid.
		Bit 04	Satellite position ephemeris data recommended refresh interval. If the accessory's long term ephemeris degrades over time, this indicates the recommended amount of time before a new download is needed.
		Bits 63:05	Reserved; set to 0.

4.13.8 Command 0x03: GetAccessoryControl

Lingo: 0x0E — Origin: Apple device

This command is sent by the Apple device to get the accessory's control state. In response, the accessory must send a `RetAccessoryControl` command within 500 ms, passing the same transaction ID and location type plus the accessory's control state information. A control type is supported only if the accessory's most recent `RetAccessoryCaps` command set the `locCapsMask` capability bit. If the accessory receives an unsupported control type, it must return an `AccessoryAck` command with a bad parameter (0x04) status.

Table 4-276 GetAccessoryControl packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
NN	1	`locType`: The location control type; see Table 4-271 (page 376).
Packet footer specified in "Command Packets" (page 109).		

4.13.9 Command 0x04: RetAccessoryControl

Lingo: 0x0E — Origin: Accessory

This command is sent by the accessory in response to a `GetAccessoryControl` command received from the Apple device. The transaction ID and control type received from `GetAccessoryControl` must be returned with this command, along with the accessory's current control state of the specified control type. System control support is required for every accessory.

Note: The `RetAccessoryControl ctlData` fields may be extended in the future as new features are added. Apple devices receiving this command check for a minimum packet payload length (rather than an exact payload length) and parse only the supported fields, ignoring any unrecognized additional data that may be appended.

Table 4-277 RetAccessoryControl packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`locType`: The location control type; see Table 4-271 (page 376).
0x*NN*	*NN*	`ctlData`: System control data, depending on the value of `locType`; see Table 4-279 (page 381).
Packet footer specified in "Command Packets" (page 109).		

4.13.10 Command 0x05: SetAccessoryControl

Lingo: 0x0E — Origin: Apple device

This command is sent by the Apple device to set the accessory's control state. In response, the accessory must send an `AccessoryAck` command within 500 ms, passing the same transaction ID and the status of the operation. Control types are valid only if the accessory's most recent `RetAccessoryCaps` command set the associated `RetAccessoryCaps locCapsMask` location type bits. If the Apple device tries to set reserved or unsupported control types or data, the accessory must return an `AccessoryAck` command with the bad parameter (0x04) status.

Accessories that support the Location lingo may enter Intermittent High Power mode as specified in the section "Accessory Power Policy" in *MFi Accessory Hardware Specification*. They may then draw high power after receiving this command with a `ctlData` value specifying GPS radio power on; see Table 4-279 (page 381). Accessories must reduce their power consumption to low power mode, as defined in *MFi Accessory Hardware Specification*, within 1 second of receiving this command with a `ctlData` value specifying GPS radio power off. Regardless of the state of power set by the Location lingo, accessories must comply with Apple device state changes as detailed in *MFi Accessory Hardware Specification*.

Note: The `SetAccessoryControl ctlData` fields may be extended in the future as new features are added. Apple devices implementing the Location lingo check for a minimum packet payload length (rather than an exact payload length) and parse only the supported fields, ignoring any unrecognized additional data that may be appended.

Table 4-278 SetAccessoryControl packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`locType`: The location control type; see Table 4-271 (page 376).
0x*NN*	*NN*	`ctlData`: The accessory controls to be set; see Table 4-279 (page 381).
Packet footer specified in "Command Packets" (page 109).		

Table 4-279 ctlData values

`locType`	Name	Bits	Meaning
0x00	`sysCtl`	System control data:	
		Bits 01:00	Accessory GPS radio power and notify control state:
			00: Power off; if the accessory is powered solely by the Apple device, it must turn its GPS radio power off while this control state is set and reduce its current consumption to low power mode within 1 second of receiving `SetAccessoryControl`.
			01–10: Reserved
			11: Power on; the accessory must turn its GPS radio power on when this control state is set. If the accessory is using Apple device power, it may draw up to 100 mA after receiving `SetAccessoryControl`.
		Bit 02	1 = Asynchronous location notifications enabled, 0 = notifications disabled.
		Bits 63:03	Reserved; set to 0.
0x01	`nmeaGpsCtrl`	Bit 00	When set to 1, NMEA GPS sentence filtering is enabled. When set to 0, NMEA GPS filtering is disabled; if asynchronous location notifications are enabled, all sentences must be passed to the Apple device.
		Bits 63:01	Reserved; set to 0.
0x02–0x3F	Reserved		
0x40–0xFF	Invalid		

4.13.11 Command 0x06: GetAccessoryData

Lingo: 0x0E — Origin: Apple device

This command is sent by the Apple device to get location data of a specific type from the accessory. In response, the accessory sends a `RetAccessoryData` command with the same transaction ID, location type, and the requested location data information. The accessory must send the corresponding `RetAccessoryData` command within 500 ms after receiving a `GetAccessoryData` command from the Apple device. In the case of multi-section responses, accessories must send each subsequent section within 500 ms of sending the previous section.

Table 4-280 GetAccessoryData packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`locType`: The capability type; see Table 4-271 (page 376).
0x*NN*	1	`dataType`: Location data type; See Table 4-281 (page 382).
Packet footer specified in "Command Packets" (page 109).		

Table 4-281 GetAccessoryData dataType values

`locType`	Name	Data type	Meaning
0x00–0x01	Reserved		
0x02	`locAsstData`	Location assistance data; this data is available only if the accessory's most recent `RetAccessoryCaps` command set the `locCapsMask` capability bit 01. The specific location data types are valid only if the associated `RetAccessoryCaps locAssistance` bits were also set.	
		0x00–0x02	Reserved
		0x03	Satellite ephemeris data maximum required refresh interval. If the accessory's long-term ephemeris degrades over time, this indicates the maximum amount of time during which downloaded data will be valid.
		0x04	Satellite ephemeris data recommended refresh interval. If the accessory's long-term ephemeris degrades over time, this indicates the recommended amount of time before a new download is needed.
		0x05–0xFF	Reserved
0x03–0x3F	Reserved		
0x40–0xFF	Invalid		

4.13.12 Command 0x07: RetAccessoryData

Lingo: 0x0E — Origin: Accessory

This command is sent by the accessory in response to a `GetAccessoryData` command received from the Apple device, returning its transaction ID, `locType`, and `dataType`, indicating the data type requested. The `locData` field is variable in length, based on the location data types listed in Table 4-281 (page 382). The Apple device acknowledges `RetAccessoryData` with an `iPodAck` command.

The maximum input data payload size that the Apple device can accept is determined by calling `RequestTransportMaxPayloadSize`. If the `locData` value sent by the accessory exceeds this size, it must be split into multiple sections as described in "Multisection Data Transfers" (page 116).

Table 4-282 RetAccessoryData packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`locType`: The capability type; see Table 4-283 (page 383).
0x*NN*	1	`dataType`: Location data type; see Table 4-283 (page 383).
0x*NN*	2	`sectCur`: Current payload section index (0x0000 = first or only section)
0x*NN*	2	`sectMax`: Maximum payload section index (0x0000 = only section, 0x*NNNN* = maximum or last section)
0x*NN*	4	`totSize`: Total size of `locData` in bytes. If multiple `RetAccessoryData` packets are sent, the `totSize` field is included only in the first packet (`sectCur` = 0x0000).
0x*NN*	*NN*	`locData`: See Table 4-283 (page 383).
Packet footer specified in "Command Packets" (page 109).		

Table 4-283 RetAccessoryData locData values

`locType`	Name	Meaning
0x00–0x01	Reserved	
0x02	`locAsstData`	Location assistance data: see Table 4-284 (page 384). This data type is valid only if the accessory's most recent `RetAccessoryCaps` command set the `sysCapsMask` capability bit 01.
0x03–0x3F	Reserved	
0x40–0xFF	Invalid	

Table 4-284 locAsstData values (locType = 0x02) for RetAccessoryData

dataType	Bytes	Field	Bytes	Subfield	Bytes	Value
0x00–0x02	Reserved.					
0x03	4	Satellite position ephemeris data maximum refresh interval, in milliseconds. Ephemeris data must be refreshed at least as often as this interval. This field must be sent in a single packet (sectCur = 0x0000 and sectMax = 0x0000).				
0x04	4	Satellite position ephemeris data recommended refresh interval, in milliseconds. Ephemeris data should be refreshed at least as often as this interval. This field must be sent in a single packet (sectCur = 0x0000 and sectMax = 0x0000). Its value must be less than or equal to the maximum refresh interval (dataType 0x03).				
0x05–0x3F	Reserved.					
0x40–0xFF	Invalid.					

4.13.13 Command 0x08: SetAccessoryData

Lingo: 0x0E — Origin: Apple device

This command is sent by the Apple device to set location data on the accessory. In response, the accessory must return an AccessoryAck command within 500 ms with the received transaction ID and the status of this operation. This time limit applies to both single and multisection responses. The dataType field represents the data type to be set; only certain data types, listed in Table 4-286 (page 385), can be set. The locData field is variable in length, based on the location data types listed in Table 4-288 (page 385).

The accessory should return its maximum payload size in its IDPS accInfo token; see Table 3-73 (page 165). If the accessory does not specify a maximum payload size, the Apple device assumes a default size of 1024 bytes. If the locData value to be sent by the Apple device exceeds this size, it will be split into multiple sections as described in "Multisection Data Transfers" (page 116).

Table 4-285 SetAccessoryData packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	locType: The capability type; see Table 4-286 (page 385).
0x*NN*	1	dataType: the data types that can be set are listed in Table 4-286 (page 385).
0x*NN*	2	sectCur: Current payload section index (0x0000 = first or only section)
0x*NN*	2	sectMax: Maximum payload section index (0x0000 = only section, 0x*NNNN* = maximum or last section)
0x*NN*	1	totSize: Total size of locData in bytes. If multiple SetAccessoryData packets are sent, the totSize field is included only in the first packet (sectCur = 0x0000).
0x*NN*	*NN*	locData: see Table 4-288 (page 385).

Value	Bytes	Parameter
Packet footer specified in "Command Packets" (page 109).		

Table 4-286 Data types settable by SetAccessoryData

locType	dataType	Description
0x01	0x00	NMEA sentence filter list string; see Table 4-287 (page 385).
0x02	0x00	Current point location data; see Table 4-288 (page 385).
0x02	0x01	Satellite ephemeris data; see Table 4-288 (page 385).
0x02	0x05	Current GPS time on accessory; see Table 4-288 (page 385).

Table 4-287 nmeaGpsLocData values (locType = 0x01)

dataType	Value
0x00	NMEA sentence filter list string, a comma-delimited set of NMEA standard and/or proprietary uppercase acronyms for sentence types. The accessory must support a filter string of at least 128 bytes. Depending on the string length, the Apple device may split this command into multiple sections. Only the last section will include the string null terminator. An example sentence list string could be `GPGGA,GPRMC`, specifying the two sentence types to be enabled. The accessory must not send sentence types not in the filter list. A string consisting of only the null terminator disables the sending of all NMEA sentences by the accessory.
0x01–0xFF	Reserved

Table 4-288 locAsstData values (locType = 0x02) for SetAccessoryData

dataType	Bytes	Field	Bytes	Subfield	Bytes	Value
0x00	16	Current point location data; must be sent as a single packet (`sectCur` = 0x0000 and `sectMax` = 0x0000). See **Note,** below.				
		`locWeek`	2	GPS Week number. The 13 least significant bits comprise a modulo-8192 representation of the current GPS Week number, as defined by IS-GPS-200 PIRN-002, Section 30.3.3.1.1.1. The GPS Week Number count began at UTC midnight Saturday-Sunday January 5-6, 1980. Since then the count has been broadcast as part of the GPS message and has incremented by 1 each UTC Saturday-to-Sunday transition. The GPS Week number will roll over in the year 2137. The week containing Tuesday, April 28, 2009 is GPS Week 1,529.		

dataType	Bytes	Field	Bytes	Subfield	Bytes	Value
		locTime	4	GPS Time of Week. The 30 least significant bits represent the time when the location point was determined. This is measured as milliseconds in the week from the last UTC Saturday-to-Sunday transition, a number between 0 and 604,800,000.		
		locPoint	8	Current location point in millionths of a degree latitude and longitude		
				latDegArc	4	Signed 32-bit latitude: 0x00000000 = ±0° (Equator), 0x055D4A80 = +90° (maximum north), 0xFAA2B580 = –90° (maximum south).
				lonDegArc	4	Signed 32-bit longitude: 0x00000000 = ±0° (Greenwich meridian), 0x0ABA9500 = +180.000000° (maximum east), 0xF5456B01 = –179.999999° (maximum west).
		locAccuracy	2	Location accuracy radius		
				locRadius	2	Point location accuracy radius in meters; a smaller radius number means a more accurate point location
0x01	<*var*>	ephData	<*var*>	Satellite ephemeris data. This data does not have a standard format and is not parsed by the Location lingo.		
0x02–0x04	Reserved.					
0x05	6	The Apple device's current GPS time.				
		locWeek	2	GPS Week number. The 13 least significant bits comprise a modulo-8192 representation of the Apple device's current GPS Week number, as defined by IS-GPS-200 PIRN-002, Section 30.3.3.1.1.1. The GPS Week Number count began at UTC midnight Saturday-Sunday January 5-6, 1980. Since then the count has been broadcast as part of the GPS message and has incremented by 1 each UTC Saturday-to-Sunday transition. The GPS Week number will roll over in the year 2137. The week containing Tuesday, April 28, 2009 is GPS Week 1,529.		
		locTime	4	GPS Time of Week. The 30 least significant bits represent the Apple device's current GPS time. This is measured as milliseconds in the week from the last UTC Saturday-to-Sunday transition, a number between 0 and 604,800,000.		
0x06–0x3F	Reserved.					
0x40–0xFF	Invalid.					

Note: The Apple device pushes system GPS time and point location data to the accessory, to be used in conjunction with ephemeris data. The GPS time of the Apple device's point location data may be significantly behind system time, because it represents the last location lock that the Apple device was able to achieve. When calculating its reference into ephemeris data, the accessory should consider both the system time and the point location time to validate the Apple device's data.

The Apple device pushes system time to the accessory when launching location services or in response to accessory inquiries. Once ephemeris data has been requested the Apple device continues to push point location data asynchronously, as the data accuracy changes. It also delivers updated ephemeris data periodically, as available, after the accessory makes its initial request.

4.13.14 Command 0x09: AsyncAccessoryData

Lingo: 0x0E — Origin: Accessory

This command is sent by the accessory to notify the Apple device that location data is available. It is valid only if the accessory supports asynchronous notifications (see Table 4-279 (page 381)) and the notification category has been enabled by a `SetAccessoryControl` command. The `dataType` field represents the data type being sent; the currently valid types are listed in Table 4-290 (page 388). The `locData` field is variable in length, depending on the data being sent.

No iAP command's length-of-packet-payload field may exceed the maximum value determined by calling `RequestTransportMaxPayloadSize`. If the `locData` value sent by the accessory exceeds this size, it must be split into multiple sections with the same transaction ID, as described in "Multisection Data Transfers" (page 116). The Apple device acknowledges each `AsyncAccessoryData` packet with an `iPodAck` command.

The accessory may send an `AsyncAccessoryData` command with a `dataType` of 0x02 or 0x05 any time Accessory Power is high (pin 13 of the 30-pin connector), regardless of the Apple device's control state as set by `SetAccessoryControl`. The Apple device responds immediately with an `iPodAck` command, but the data requested will not be sent until the Apple device's location services are actually engaged by an application. The accessory must respond to other iAP commands while waiting for a response. The URL must not be sent to the Apple device more often than the maximum ephemeris refresh interval established by the accessory via `RetAccessoryData`, and in any case not more often than once every 5 minutes.

Table 4-289 AsyncAccessoryData packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`locType`: The capability type; see Table 4-290 (page 388).
0x*NN*	1	`dataType`: Location data type; see Table 4-290 (page 388).
0x*NN*	2	`sectCur`: Current payload section index (0x0000 = first or only section)
0x*NN*	2	`sectMax`: Maximum payload section index (0x0000 = only section, 0x*NNNN* = maximum or last section)
0x*NN*	4	`totSize`: Total size of `locData` in bytes. If multiple `AsyncAccessoryData` packets are sent, the `totSize` field is included only in the first packet (`sectCur` = 0x0000). See "Multisection Data Transfers" (page 116).

Value	Bytes	Parameter
0x*NN*	*NN*	`locData`: Data sent by `AsyncAccessoryData`; see Table 4-290 (page 388).
Packet footer specified in "Command Packets" (page 109).		

Table 4-290 locData values that can be sent by AsyncAccessoryData

`locType`	`dataType`	`locData` **value**
0x01	0x80	String of NMEA 0183 compliant sentences. Each sentence must be sent as a full NMEA 0183 compliant sentence including the `$GP`, `$P`, or `!P` prefix, the `*NN` checksum, and the `<CR><LF>` suffix.
0x02	0x02	Null-terminated URL string (RFC 1738 compliant) to a web site for satellite ephemeris data (such as `http://www.yourcompany.com/long-term-ephemeris`). This must be a full URL, including `http://`. Only http and https transfer protocols are currently supported.
0x02	0x05	Request for current system time in GPS time (GPS Week number and milliseconds offset into the week; see Table 4-288 (page 385)).

4.13.15 Command 0x80: iPodAck

Lingo: 0x0E — Origin: Apple device

This command is sent by the Apple device in response to a command sent from the accessory. It has two forms:

- The Default form, shown in Table 4-291 (page 388), is sent under these conditions:
  - When a bad parameter is received, an unsupported or invalid command is received, or a command that does not return any data has completed.
  - To indicate completion of a multisection data transfer, as described in "Multisection Data Transfers" (page 116).
- The Section form is sent if the command from the accessory is part of a multisection data transfer. This form of the command is discussed in "Multisection iPodAck Command" (page 389).

Table 4-291 Default iPodAck packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command status; see Table 3-6 (page 125).
0x*NN*	1	`cmdIDOrig`: Original command ID for which this response is being sent.
Packet footer specified in "Command Packets" (page 109).		

4.13.15.1 Multisection iPodAck Command

Lingo: 0x0E — Origin: Apple device

An `iPodAck` command is sent by the Apple device in response to each multisection data transfer packet sent by `RetAccessoryData` or `AsyncAccessoryData`. The Apple device may send either of two forms of the `iPodAck` packet:

- When any section before the last section has been received successfully, the Apple device sends the packet shown in Table 4-292 (page 389). The accessory may send a new packet immediately.
- When the last section has been received successfully, the Apple device sends the packet shown in Table 4-291 (page 388) with `ackStatus` = 0x00, indicating that the multisection data transfer is complete. The Apple device also sends an acknowledgment of this form, with a nonzero `ackStatus`, when an error has occurred in receiving any packet.

Table 4-292 Multisection iPodAck packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x13	1	`ackStatus`: Section received successfully
0x*NN*	1	`cmdIDOrig`: Original command ID for which this response is being sent.
0x*NN*	2	`sectCur`: Section index for which this response is being sent.
Packet footer specified in "Command Packets" (page 109).		

4.14 Sample Identification Sequences

This section provides the following commented listings of sample command sequences in which an accessory identifies itself to an Apple device:

- Table 4-293 (page 389) lists sample commands for an accessory that supports the General and Extended Interface lingoes. For Extended Interface command details, see "Extended Interface Mode" (page 397).
- Table 4-294 (page 391) lists sample commands for an accessory that supports the General, Simple Remote, and Display Remote lingoes.
- Table 4-295 (page 393) lists sample commands for an accessory that supports the General, Simple Remote, Display Remote, and Digital Audio lingoes.

Table 4-293 Identification of lingoes 0x00+0x04

Step	Accessory command	Apple device command	Comment
1	`StartIDPS`		no params
2		`iPodAck`	acknowledging 'Success (OK)' to command 'General Lingo::StartIDPS'

Step	Accessory command	Apple device command	Comment
If the `iPodAck` reply to `StartIDPS` returns a status of 0x00, the accessory proceeds to Step 3. Otherwise, ■ If the accessory receives no `iPodAck` command within 500 ms, it may retry Step 1 at 500 ms intervals as long as Apple Device Detect (pin 30 of the 30-pin connector) is low and Accessory Power (pin 13) is high. Accessories that don't monitor these signals (such as Bluetooth accessories or USB accessories without a 191 kΩ ID resistor) may retry without checking them. ■ If the accessory receives an `iPodAck` status of 0x04 (Bad Parameter), the Apple device does not support IDPS. Within 800 ms the accessory should send an `IdentifyDeviceLingoes` command with all fields set to 0xFF. This causes the Apple device to display an "Accessory not supported" message. Alternatively, an accessory receiving a status of 0x04 may send an `IdentifyDeviceLingoes` command with its lingo mask set for only the General lingo, with no options, and with a Device ID of 0x0. See "Canceling a Current Authentication Process With IdentifyDeviceLingoes" (page 135). A sample command sequence is listed in Table C-50 (page 556). ■ If the accessory receives any other nonzero `iPodAck` status, it must not send any more iAP commands until it has been disconnected. Upon reconnection it must restart identification at Step 1.			
3	`SetFIDTokenValues`		setting 11 FID tokens ; IdentifyToken = (lingoes: 0/4 | options 0x00000002 | device ID 0x00000200); AccessoryCapsToken = 0x0000000000000205; AccessoryInfoToken = Acc name (Apple); AccessoryInfoToken = Acc FW version (v9.8.7); AccessoryInfoToken = Acc HW version (v1.2.3); AccessoryInfoToken = Acc manufacturer (Apple Inc.); AccessoryInfoToken = Acc model number (ModelNumber-XYZ); EAProtocolToken = 1 (com.Apple.ProtocolMain); EAProtocolToken = 2 (com.Apple.ProtocolAlt); iPodPreferenceToken = (setting 'off' for preference class 'video out setting' with restore on exit 'not selected'); iPodPreferenceToken = (setting 'used' for preference class 'line out usage' with restore on exit 'not selected')

Step	Accessory command	Apple device command	Comment
4		AckFIDTokenValues	11 ACKs for FID tokens ; IdentifyToken = accepted; AccessoryCapsToken = accepted; AccessoryInfoToken = (Acc name) accepted; AccessoryInfoToken = (Acc FW version) accepted; AccessoryInfoToken = (Acc HW version) accepted; AccessoryInfoToken = (Acc manufacturer) accepted; AccessoryInfoToken = (Acc model number) accepted; EAProtocolToken = (1) accepted; EAProtocolToken = (2) accepted; iPodPreferenceToken = (video out setting) accepted; iPodPreferenceToken = (line out usage) accepted
5	EndIDPS		status 'finished with IDPS; proceed to authentication'
6		IDPSStatus	status 'ready for auth'
7		GetAccessory-AuthenticationInfo	no params

Table 4-294 Identification of lingoes 0x00+0x02+0x03

Step	Accessory command	Apple device command	Comment
1	StartIDPS		no params
2		iPodAck	acknowledging 'Success (OK)' to command 'General Lingo::StartIDPS'

If the iPodAck reply to StartIDPS returns a status of 0x00, the accessory proceeds to Step 3. Otherwise,

- If the accessory receives no iPodAck command within 500 ms, it may retry Step 1 at 500 ms intervals as long as Apple Device Detect (pin 30 of the 30-pin connector) is low and Accessory Power (pin 13) is high. Accessories that don't monitor these signals (such as Bluetooth accessories or USB accessories without a 191 kΩ ID resistor) may retry without checking them.
- If the accessory receives an iPodAck status of 0x04 (Bad Parameter), the Apple device does not support IDPS. Within 800 ms the accessory should send an IdentifyDeviceLingoes command with all fields set to 0xFF. This causes the Apple device to display an "Accessory not supported" message.

  Alternatively, an accessory receiving a status of 0x04 may send an IdentifyDeviceLingoes command with its lingo mask set for only the General lingo, with no options, and with a Device ID of 0x0. See "Canceling a Current Authentication Process With IdentifyDeviceLingoes" (page 135). A sample command sequence is listed in Table C-52 (page 565).
- If the accessory receives any other nonzero iPodAck status, it must not send any more iAP commands until it has been disconnected. Upon reconnection it must restart identification at Step 1.

Step	Accessory command	Apple device command	Comment
3	SetFIDTokenValues		setting 11 FID tokens ; IdentifyToken = (lingoes: 0/2/3 | options 0x00000002 | device ID 0x00000200); AccessoryCapsToken = 0x0000000000000205; AccessoryInfoToken = Acc name (Apple); AccessoryInfoToken = Acc FW version (v9.8.7); AccessoryInfoToken = Acc HW version (v1.2.3); AccessoryInfoToken = Acc manufacturer (Apple Inc.); AccessoryInfoToken = Acc model number (ModelNumber-XYZ); EAProtocolToken = 1 (com.Apple.ProtocolMain); EAProtocolToken = 2 (com.Apple.ProtocolAlt); iPodPreferenceToken = (setting 'off' for preference class 'video out setting' with restore on exit 'not selected'); iPodPreferenceToken = (setting 'used' for preference class 'line out usage' with restore on exit 'not selected')
4		AckFIDTokenValues	11 ACKs for FID tokens ; IdentifyToken = accepted; AccessoryCapsToken = accepted; AccessoryInfoToken = (Acc name) accepted; AccessoryInfoToken = (Acc FW version) accepted; AccessoryInfoToken = (Acc HW version) accepted; AccessoryInfoToken = (Acc manufacturer) accepted; AccessoryInfoToken = (Acc model number) accepted; EAProtocolToken = (1) accepted; EAProtocolToken = (2) accepted; iPodPreferenceToken = (video out setting) accepted; iPodPreferenceToken = (line out usage) accepted
5	EndIDPS		status 'finished with IDPS; proceed to authentication'
6		IDPSStatus	status 'ready for auth'
7		GetAccessory-AuthenticationInfo	no params

Table 4-295 Identification of lingoes 0x00+0x02+0x03+0x0A

Step	Accessory command	Apple device command	Comment
1	`StartIDPS`		no params
2		`iPodAck`	acknowledging 'Success (OK)' to command 'General Lingo::StartIDPS'
If the `iPodAck` reply to `StartIDPS` returns a status of 0x00, the accessory proceeds to Step 3. Otherwise, ■ If the accessory receives no `iPodAck` command within 500 ms, it may retry Step 1 at 500 ms intervals as long as Apple Device Detect (pin 30 of the 30-pin connector) is low and Accessory Power (pin 13) is high. Accessories that don't monitor these signals (such as Bluetooth accessories or USB accessories without a 191 kΩ ID resistor) may retry without checking them. ■ If the accessory receives an `iPodAck` status of 0x04 (Bad Parameter), the Apple device does not support IDPS. Within 800 ms the accessory should send an `IdentifyDeviceLingoes` command with all fields set to 0xFF. This causes the Apple device to display an "Accessory not supported" message. Alternatively, an accessory receiving a status of 0x04 may send an `IdentifyDeviceLingoes` command with its lingo mask set for only the General lingo, with no options, and with a Device ID of 0x0. See "Canceling a Current Authentication Process With IdentifyDeviceLingoes" (page 135). A sample command sequence is listed in Table C-53 (page 568). ■ If the accessory receives any other nonzero `iPodAck` status, it must not send any more iAP commands until it has been disconnected. Upon reconnection it must restart identification at Step 1.			
3	`SetFIDTokenValues`		setting 11 FID tokens ; IdentifyToken = (lingoes: 0/2/3/10 | options 0x00000002 | device ID 0x00000200); AccessoryCapsToken = 0x0000000000000215; AccessoryInfoToken = Acc name (Apple); AccessoryInfoToken = Acc FW version (v9.8.7); AccessoryInfoToken = Acc HW version (v1.2.3); AccessoryInfoToken = Acc manufacturer (Apple Inc.); AccessoryInfoToken = Acc model number (ModelNumber-XYZ); EAProtocolToken = 1 (com.Apple.ProtocolMain); EAProtocolToken = 2 (com.Apple.ProtocolAlt); iPodPreferenceToken = (setting 'off' for preference class 'video out setting' with restore on exit 'not selected'); iPodPreferenceToken = (setting 'used' for preference class 'line out usage' with restore on exit 'not selected')

Step	Accessory command	Apple device command	Comment
4		AckFIDTokenValues	11 ACKs for FID tokens ; IdentifyToken = accepted; AccessoryCapsToken = accepted; AccessoryInfoToken = (Acc name) accepted; AccessoryInfoToken = (Acc FW version) accepted; AccessoryInfoToken = (Acc HW version) accepted; AccessoryInfoToken = (Acc manufacturer) accepted; AccessoryInfoToken = (Acc model number) accepted; EAProtocolToken = (1) accepted; EAProtocolToken = (2) accepted; iPodPreferenceToken = (video out setting) accepted; iPodPreferenceToken = (line out usage) accepted
5	EndIDPS		status 'finished with IDPS; proceed to authentication'
6		IDPSStatus	status 'ready for auth'
7		GetAccessory-AuthenticationInfo	no params
8	RetAccessory-AuthenticationInfo		returns authentication version and X.509 certificate data
9		AckAccessory-AuthenticationInfo	acknowledging 'auth info supported'
10		GetAccessorySample-RateCaps	no params
11		GetAccessory-Authentication-Signature	sends authentication challenge
12	RetAccessorySample-RateCaps		returning sample rates '8000 | 11025 | 12000 | 16000 | 22050 | 24000 | 32000 | 44100 | 48000'
13	RetAccessory-Authentication-Signature		returns digital signature in response to authentication challenge
14		TrackNewAudio-Attributes	sample rate 44100; Sound Check value 0; track volume adjustment 0
15	AccessoryAck		acknowledging 'TrackNewAudioAttributes'

Step	Accessory command	Apple device command	Comment
16		AckAccessory-AuthenticationStatus	acknowledging authentication status 'Success (OK)'
17		TrackNewAudio-Attributes	sample rate 44100; Sound Check value 0; track volume adjustment 0
18	AccessoryAck		acknowledging 'TrackNewAudioAttributes'
19		TrackNewAudio-Attributes	sample rate 44100; Sound Check value 0; track volume adjustment 0
20	AccessoryAck		acknowledging 'TrackNewAudioAttributes'

5. Extended Interface Mode

This chapter specifies Extended Interface mode, an operating mode of Apple devices that allows their user interfaces to be translated to other environments in attached accessories.

For the purpose of this document, Apple devices can be considered to operate in two major modes: Standard UI mode and Extended Interface mode. For more information on Apple device power states, see *MFi Accessory Hardware Specification*.

Standard UI Mode is the user interface mode that allows an Apple device to be driven by its front panel display and buttons. Alternatively, the Apple device may be placed in Extended Interface mode and controlled by an accessory, using the Extended Interface mode lingo (Lingo 0x04) described in this chapter.

The Apple device transitions into the Extended Interface mode when an accessory is connected to it and issues a `SetUIMode` command. The transition to Extended Interface mode is described in "Entering Extended Interface Mode" (page 57).

If the Apple device is playing an audio track during this transition, the playback is automatically paused; video tracks are stopped. By default, the Apple device's display changes to an "OK to disconnect" screen such as one of those shown in Figure 5-1 (page 397).

Figure 5-1 Typical "OK to disconnect" screens

Extended Interface mode allows accessories to replace the checkmark graphic with a downloaded image set through "Command 0x0032: SetDisplayImage" (page 435). Removing power from the Apple device while a connection remains results in the Apple device going into a Sleep state after two minutes of inactivity. The keys and wheel on the front of the Apple device are disabled when in Extended Interface mode.

The Apple device transitions back to Standard UI mode when any of the following occurs:

- The accessory issues an `ExitExtendedInterfaceMode` command.

- The accessory reidentifies itself, using the `StartIDPS` command (see "Transport Initialization" (page 88)).
- The accessory is disconnected from the Apple device.

If the Apple device is playing a track during this transition, the playback is automatically paused. Any Apple device settings with the restore on exit feature set are restored when the Apple device is disconnected.

5.1 Command Packets

This section describes the individual Extended Interface (Lingo 0x04) command packets.

5.1.1 Command Code Summary

Table 5-1 (page 398) lists the valid command codes for Extended Interface mode.

Table 5-1 Extended Interface lingo command summary

Command ID	Command	Target engine	Introduced in protocol version	Requires authentication over serial transport
0x0000	Reserved	N/A	N/A	N/A
0x0001	`iPodAck`	N/A	1.00	No
0x0002	`GetCurrentPlaying-TrackChapterInfo`	Playback Engine	1.06	No
0x0003	`ReturnCurrentPlaying-TrackChapterInfo`	Playback Engine	1.06	No
0x0004	`SetCurrentPlaying-TrackChapter`	Playback Engine	1.06	No
0x0005	`GetCurrentPlaying-TrackChapterPlayStatus`	Playback Engine	1.06	No
0x0006	`ReturnCurrentPlaying-TrackChapterPlayStatus`	Playback Engine	1.06	No
0x0007	`GetCurrentPlaying-TrackChapterName`	Playback Engine	1.06	No
0x0008	`ReturnCurrentPlaying-TrackChapterName`	Playback Engine	1.06	No
0x0009	`GetAudiobookSpeed`	N/A	1.06	No

Command ID	Command	Target engine	Introduced in protocol version	Requires authentication over serial transport
0x000A	ReturnAudiobookSpeed	N/A	1.06	No
0x000B	SetAudiobookSpeed	N/A	1.06	No
0x000C	GetIndexedPlaying-TrackInfo	Playback Engine	1.08	No
0x000D	ReturnIndexedPlaying-TrackInfo	N/A	1.08	No
0x000E	GetArtworkFormats	Playback Engine	1.10	No
0x000F	RetArtworkFormats	Playback Engine	1.10	No
0x0010	GetTrackArtworkData	Playback Engine	1.10	No
0x0011	RetTrackArtworkData	Playback Engine	1.10	No
0x0012	RequestProtocolVersion	See "Deprecated Extended Interface Commands" (page 543).		
0x0013	ReturnProtocolVersion	See "Deprecated Extended Interface Commands" (page 543).		
0x0014	RequestiPodName	See "Deprecated Extended Interface Commands" (page 543).		
0x0015	ReturniPodName	See "Deprecated Extended Interface Commands" (page 543).		
0x0016	ResetDBSelection	Database Engine	1.00	No
0x0017	SelectDBRecord	Database Engine As an optimization, selecting a single track or audiobook automatically passes it to the player.	1.00	No
0x0018	GetNumberCategorized-DBRecords	Database Engine	1.00	No
0x0019	ReturnNumber-CategorizedDBRecords	N/A	1.00	No
0x001A	RetrieveCategorized-DatabaseRecords	Database Engine	1.00	No

Command ID	Command	Target engine	Introduced in protocol version	Requires authentication over serial transport
0x001B	ReturnCategorized-DatabaseRecord	N/A	1.00	No
0x001C	GetPlayStatus	Playback Engine	1.00	No
0x001D	ReturnPlayStatus	N/A	1.00	No
0x001E	GetCurrentPlaying-TrackIndex	Playback Engine	1.00	No
0x001F	ReturnCurrentPlaying-TrackIndex	N/A	1.00	No
0x0020	GetIndexedPlaying-TrackTitle	Playback Engine	1.00	No
0x0021	ReturnIndexedPlaying-TrackTitle	N/A	1.00	No
0x0022	GetIndexedPlaying-TrackArtistName	Playback Engine	1.00	No
0x0023	ReturnIndexedPlaying-TrackArtistName	N/A	1.00	No
0x0024	GetIndexedPlaying-TrackAlbumName	Playback Engine	1.00	No
0x0025	ReturnIndexedPlaying-TrackAlbumName	N/A	1.00	No
0x0026	SetPlayStatusChange-Notification	Playback Engine	1.00	No
0x0027	PlayStatusChange-Notification	N/A	1.00	No
0x0028	PlayCurrentSelection	**Deprecated;** see "Command 0x0028: PlayCurrentSelection" (page 545). Do not use this command in new accessory designs; use "Command 0x0017: SelectDBRecord" (page 415) instead.		
0x0029	PlayControl	Playback Engine	1.00	No
0x002A	GetTrackArtworkTimes	Playback Engine	1.10	No
0x002B	RetTrackArtworkTimes	Playback Engine	1.10	No
0x002C	GetShuffle	N/A	1.00	No

Command ID	Command	Target engine	Introduced in protocol version	Requires authentication over serial transport
0x002D	ReturnShuffle	N/A	1.00	No
0x002E	SetShuffle	N/A	1.00	No
0x002F	GetRepeat	N/A	1.00	No
0x0030	ReturnRepeat	N/A	1.00	No
0x0031	SetRepeat	N/A	1.00	No
0x0032	SetDisplayImage	N/A	1.01	No
0x0033	GetMonoDisplay-ImageLimits	N/A	1.01	No
0x0034	ReturnMonoDisplay-ImageLimits	N/A	1.01	No
0x0035	GetNumPlayingTracks	Playback Engine	1.01	No
0x0036	ReturnNumPlayingTracks	N/A	1.01	No
0x0037	SetCurrentPlayingTrack	Playback Engine	1.01	No
0x0038	SelectSortDBRecord	**Deprecated;** do not use in new accessory designs. See "Command 0x0038: SelectSortDBRecord" (page 546).		
0x0039	GetColorDisplay-ImageLimits	N/A	1.09	No
0x003A	ReturnColorDisplay-ImageLimits	N/A	1.09	No
0x003B	ResetDBSelection-Hierarchy	Database Engine	1.11	Yes
0x003C	GetDBiTunesInfo	Database Engine	1.13	Yes
0x003D	RetDBiTunesInfo	Database Engine	1.13	Yes
0x003E	GetUIDTrackInfo	Database Engine	1.13	Yes
0x003F	RetUIDTrackInfo	Database Engine	1.13	Yes
0x0040	GetDBTrackInfo	Database Engine	1.13	Yes
0x0041	RetDBTrackInfo	Database Engine	1.13	Yes
0x0042	GetPBTrackInfo	Playback Engine	1.13	Yes

Command ID	Command	Target engine	Introduced in protocol version	Requires authentication over serial transport
0x0043	RetPBTrackInfo	Playback Engine	1.13	Yes
0x0044	CreateGeniusPlaylist	Database Engine and Playback Engine	1.13	Yes
0x0045	RefreshGeniusPlaylist	Database Engine	1.13	Yes
0x0046	Reserved	N/A	N/A	N/A
0x0047	IsGeniusAvailable-ForTrack	Database Engine and Playback Engine	1.13	Yes
0x0048	GetPlaylistInfo	Database Engine	1.13	Yes
0x0049	RetPlaylistInfo	Database Engine	1.13	Yes
0x004A	PrepareUIDList	Playback Engine	1.14	Yes
0x004B	PlayPreparedUIDList	Playback Engine	1.14	Yes
0x004C	GetArtworkTimes	Playback Engine	1.14	Yes
0x004D	RetArtworkTimes	Playback Engine	1.14	Yes
0x004E	GetArtworkData	Playback Engine	1.14	Yes
0x004F	RetArtworkData	Playback Engine	1.14	Yes
0x0050 – 0xFFFF	Reserved	N/A	N/A	N/A

5.1.2 Command 0x0001: iPodAck

Lingo: 0x04 — Origin: Apple device

The Apple device sends this command to acknowledge the receipt of a command and return the command status. The command ID field indicates the accessory command for which the response is being sent. The command status indicates the results of the command (success or failure).

Table 5-2 iPodAck packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command result status; see Table 3-6 (page 125).
0x*NN*	2	The ID of the command being acknowledged.

Value	Bytes	Parameter
Packet footer specified in "Command Packets" (page 109).		

5.1.3 Command 0x0002: GetCurrentPlayingTrackChapterInfo

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

Requests the chapter information of the currently playing track. In response, the Apple device sends a "Command 0x0003: ReturnCurrentPlayingTrackChapterInfo" (page 403) command to the accessory.

Note: The returned track index is valid only when there is a currently playing or paused track.

Table 5-3 `GetCurrentPlayingTrackChapterInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

5.1.4 Command 0x0003: ReturnCurrentPlayingTrackChapterInfo

Lingo: 0x04 — Origin: Apple device

Returns the chapter information of the currently playing track. The Apple device sends this command in response to the "Command 0x0002: GetCurrentPlayingTrackChapterInfo" (page 403) command from the accessory. The track chapter information includes the currently playing track's chapter index, as well as the total number of chapters in the track. The track chapter and the total number of chapters are 32-bit signed integers. The chapter index of the first chapter is always 0x00000000. If the track does not have chapter information, a chapter index of –1 (0xFFFFFFFF) and a chapter count of 0 (0x00000000) are returned.

Table 5-4 `ReturnCurrentPlayingTrackChapterInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Current chapter index
0x*NN*	4	Chapter count
Packet footer specified in "Command Packets" (page 109).		

5.1.5 Command 0x0004: SetCurrentPlayingTrackChapter

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

Sets the currently playing track chapter. You can send the "Command 0x0002: GetCurrentPlayingTrackChapterInfo" (page 403) command to get the chapter count and the index of the currently playing chapter in the current track. In response to the `SetCurrentPlayingTrackChapter` command, the Apple device sends an `iPodAck` command with the command status.

Note: This command should be used only when the Apple device is in a playing or paused state. The command fails if the Apple device is stopped or if the track does not contain chapter information.

Table 5-5 `SetCurrentPlayingTrackChapter` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Chapter index
Packet footer specified in "Command Packets" (page 109).		

5.1.6 Command 0x0005: GetCurrentPlayingTrackChapterPlayStatus

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

Requests the chapter playtime status of the currently playing track. The status includes the chapter length and the time elapsed within that chapter. In response to a valid command, the Apple device sends a "Command 0x0006: ReturnCurrentPlayingTrackChapterPlayStatus" (page 405) command to the accessory.

Note: If the packet length or chapter index is invalid—for instance, if the track does not contain chapter information—the Apple device responds with an `iPodAck` command including the specific error status.

Table 5-6 `GetCurrentPlayingTrackChapterPlayStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Currently playing chapter index
Packet footer specified in "Command Packets" (page 109).		

5.1.7 Command 0x0006: ReturnCurrentPlayingTrackChapterPlayStatus

Lingo: 0x04 — Origin: Apple device

Returns the play status of the currently playing track chapter. The Apple device sends this command in response to the "Command 0x0005: GetCurrentPlayingTrackChapterPlayStatus" (page 404) command from the accessory. The returned information includes the chapter length and elapsed time, in milliseconds.

Table 5-7 `ReturnCurrentPlayingTrackChapterPlayStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Chapter length in milliseconds
0x*NN*	4	Elapsed time in chapter, in milliseconds
Packet footer specified in "Command Packets" (page 109).		

5.1.8 Command 0x0007: GetCurrentPlayingTrackChapterName

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

Requests a chapter name in the currently playing track. In response to a valid command, the Apple device sends a "Command 0x0008: ReturnCurrentPlayingTrackChapterName" (page 405) command to the accessory.

Note: If the received packet length or track index is invalid—for instance, if the track does not have chapter information or is not a part of the Audiobook category—the Apple device responds with an `iPodAck` command including the specific error status.

Table 5-8 `GetCurrentPlayingTrackChapterName` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Chapter index
Packet footer specified in "Command Packets" (page 109).		

5.1.9 Command 0x0008: ReturnCurrentPlayingTrackChapterName

Lingo: 0x04 — Origin: Apple device

Returns a chapter name in the currently playing track. The Apple device sends this command in response to a valid "Command 0x0007: GetCurrentPlayingTrackChapterName" (page 405) command from the accessory. The chapter name is encoded as a null-terminated UTF-8 character array.

Note: The chapter name string is not limited to 252 characters; it may be sent in small or large packet format depending on the string length. The small packet format is shown.

Table 5-9 `ReturnCurrentPlayingTrackChapterName` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	*NN*	Chapter name as UTF-8 character array
Packet footer specified in "Command Packets" (page 109).		

5.1.10 Command 0x0009: GetAudiobookSpeed

Lingo: 0x04 — Origin: Accessory

Requests the current Apple device audiobook speed state. The Apple device responds with the "Command 0x000A: ReturnAudiobookSpeed" (page 406) command indicating the current audiobook speed.

Table 5-10 `GetAudiobookSpeed` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

5.1.11 Command 0x000A: ReturnAudiobookSpeed

Lingo: 0x04 — Origin: Apple device

Returns the current audiobook speed setting. The Apple device sends this command in response to the "Command 0x0009: GetAudiobookSpeed" (page 406) command from the accessory.

Table 5-11 `ReturnAudiobookSpeed` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Audiobook speed status code. See Table 5-12 (page 407).

Value	Bytes	Parameter
Packet footer specified in "Command Packets" (page 109).		

Table 5-12 (page 407) shows the possible audiobook speed states returned by this command.

Table 5-12 Audiobook speed states

Value	Meaning
0xFF	Slower (–1)
0x00	Normal
0x01	Faster (+1)
0x02 – 0xFE	Reserved

5.1.12 Command 0x000B: SetAudiobookSpeed

Direction: Accessory to Apple device

Sets the speed of audiobook playback. The Apple device audiobook speed states are listed in Table 5-12 (page 407). This command has two modes: one to set the speed of the currently playing audiobook and a second to set the audiobook speed for all audiobooks.

The Restore on Exit field is optional; a nonzero value restores the original audiobook speed setting when the accessory is detached. If this field is zero, the audiobook speed setting set by the accessory is saved and persists after the accessory is detached from the Apple device. If this field is omitted from the command packet, the command affects only the currently playing audiobook. In response to this command, the Apple device sends an `iPodAck` command with the command status.

Note: Accessory developers are encouraged to always include the Restore on Exit field with a nonzero value, to restore any of the user's Apple device settings that were modified by the accessory.

Table 5-13 `SetAudiobookSpeed` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	New audiobook speed code. See Table 5-12 (page 407).
0x*NN*	1	Optional Restore on Exit: If 1, the original setting is restored on detach; if 0, the new setting persists after accessory detach. If this byte is included in the command packet, the `SetAudiobookSpeed` command sets the speed of all audiobooks. If this byte is omitted, its sets the speed of only the currently playing audiobook.
Packet footer specified in "Command Packets" (page 109).		

5.1.13 Command 0x000C: GetIndexedPlayingTrackInfo

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

Gets track information for the track at the specified index. The track info type field specifies the type of information to be returned, such as current song lyrics, podcast name, episode date, and episode description. In response, the Apple device sends the "Command 0x000D: ReturnIndexedPlayingTrackInfo" (page 409) command with the requested track information. If the information type is invalid or does not apply to the selected track, the Apple device returns an `iPodAck` with an error status.

An accessory can determine the number of tracks in the list of tracks queued to play on the Apple device by sending a "Command 0x0035: GetNumPlayingTracks" (page 441) command and receiving a "Command 0x0036: ReturnNumPlayingTracks" (page 441) command in reply.

Table 5-14 `GetIndexedPlayingTrackInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Track info type. See Table 5-15 (page 408).
0x*NN*	4	Track index
0x*NN*	2	Chapter index
Packet footer specified in "Command Packets" (page 109).		

Table 5-15 (page 408) shows the types of information you can obtain for a track.

Table 5-15 Track information type

Info Type	Code
0x00	Track Capabilities and Information
0x01	Podcast Name
0x02	Track Release Date
0x03	Track Description
0x04	Track Song Lyrics
0x05	Track Genre
0x06	Track Composer
0x07	Track Artwork Count

Note: Requesting the lyrics of a track not currently being played will result in a null string being returned. Lyrics for a track being played may take up to 5 seconds to return. Use `SetPlayStatusChangeNotification` to be notified when the lyrics for the currently playing track become available.

5.1.14 Command 0x000D: ReturnIndexedPlayingTrackInfo

Direction: Apple device to Accessory

Returns the requested track information type and data. The Apple device sends this command in response to the "Command 0x000C: GetIndexedPlayingTrackInfo" (page 408) command. Table 5-18 (page 410) lists the available information types and their data.

Data returned as strings are encoded as null-terminated UTF-8 character arrays. If the track information string does not exist, a null UTF-8 string is returned. If the track has no release date, then the returned release date has all bytes zeros. Track song lyrics and the track description are sent in a large or small packet format with an incrementing packet index field, spanning multiple packets if needed.

Table 5-16 (page 409) shows the packet format for the `ReturnIndexedPlayingTrackInfo` command sent in response to a request for information types 0x00 to 0x02.

Table 5-16 `ReturnIndexedPlayingTrackInfo` packet for info types 0x00-0x02 and 0x05-0x07

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Track info type. See Table 5-18 (page 410).
0x*NN*	*NN*	Track information. The data format is specific to the track info type. See Table 5-18 (page 410).
Packet footer specified in "Command Packets" (page 109).		

Table 5-17 (page 409) shows the packet format for the `ReturnIndexedPlayingTrackInfo` command sent in response to a request for information types 0x03 to 0x04.

Table 5-17 `ReturnIndexedPlayingTrackInfo` command for info types 0x03-0x04

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Track info type. See Table 5-18 (page 410).
0x*NN*	1	Packet information bits. If set, these bits have the following meanings: ■ Bit 7:2 Reserved ■ Bit 1: This is the last packet. Applicable only if bit 0 is set. ■ Bit 0: Indicates that there are multiple packets.

Value	Bytes	Parameter
0x*NN*	2	Packet index
0x*NN*	*NN*	Track information as a UTF-8 string.
Packet footer specified in "Command Packets" (page 109).		

Table 5-18 (page 410) shows the available track information types and the format of the data returned for each type.

Table 5-18 Track info types and return data

Info Type	Code	Data Format
0x00	Track Capabilities and Information	A 10-byte data field. See Table 5-19 (page 411).
0x01	Podcast Name	UTF-8 string
0x02	Track Release Date	An 8-byte data field. See Table 5-20 (page 411)
0x03	Track Description	UTF-8 string
0x04	Track Song Lyrics	UTF-8 string
0x05	Track Genre	UTF-8 string
0x06	Track Composer	UTF-8 string
0x07	Track Artwork Count	Artwork count data. The artwork count is a sequence of 4-byte records; each record consists of a 2-byte format ID value followed by a 2-byte count of images in that format for this track. For more information about formatID and chapter index values, see commands 0x000E-0x0011 and 0x002A-0x002B.
0x08-0xFF	Reserved	N/A

Table 5-19 (page 411) shows the data returned for the Track Capabilities and Information type.

Table 5-19 Track Capabilities and Information encoding

Byte number	Code
0-3	Track Capability bits. If set, these bits have the following meanings: ■ Bits 31:15: Reserved ■ Bit 14: Track is an iTunesU episode ■ Bit 13: Track is capable of generating a Genius playlist ■ Bits 12:9: Reserved ■ Bit 8: Track is currently queued to play as a video ■ Bit 7: Track contains video (a video podcast, music video, movie, or TV show) ■ Bit 6: Track has description ■ Bit 5: Track has release date ■ Bit 4: Track is a podcast episode ■ Bit 3: Track has song lyrics ■ Bit 2: Track has album artwork ■ Bit 1: Track has chapters ■ Bit 0: Track is audiobook
4	Total track length, in milliseconds (bits 31:24)
5	Total track length, in milliseconds (bits 23:16)
6	Total track length, in milliseconds (bits 15:8)
7	Total track length, in milliseconds (bits 7:0)
8	Chapter count (bits 15:8)
9	Chapter count (bits 7:0)

Table 5-20 (page 411) shows the data returned for the Track Release Date type.

Table 5-20 Track Release Date encoding

Byte number	Code
0	Seconds (0-59)
1	Minutes (0-59)
2	Hours (0-23)
3	Day of the month (1-31)
4	Month (1–12)

Byte number	Code
5	Year (bits 15:8). For example, 2006 signifies the year 2006 AD.
6	Year (bits 7:0).
7	Weekday (0-6, where 0 = Sunday and 6 = Saturday)

5.1.15 Command 0x000E: GetArtworkFormats

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

The accessory sends this command to obtain the list of supported artwork formats on the Apple device. No parameters are sent. See "Transferring Album Art" (page 64).

Table 5-21 `GetArtworkFormats` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

5.1.16 Command 0x000F: RetArtworkFormats

Lingo: 0x04 — Origin: Apple device

Applies To: Playback Engine

The Apple device sends this command to the accessory, giving it the list of supported artwork formats. Each format is described in a 7-byte record (formatID:2, pixelFormat:1, width:2, height:2). The `formatID` is used when sending `GetArtworkTimes`. The accessory may return zero records. See "Transferring Album Art" (page 64).

Table 5-22 `RetArtworkFormats` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	2	`formatID`: Apple device-assigned value for this format.
0x*NN*	1	`pixelFormat`: Same as from `SetDisplayImage`.
0x*NN*	2	`imageWidth`: Number of pixels wide for each image.
0x*NN*	2	`imageHeight`: Number of pixels high for each image.

Value	Bytes	Parameter
0x*NN*	*NN*	Previous 7 bytes may be repeated *NN* times.
Packet footer specified in "Command Packets" (page 109).		

5.1.17 Command 0x0010: GetTrackArtworkData

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

The accessory sends this command to the Apple device to request data for a given `trackIndex`, `formatID`, and `artworkIndex`. See "Transferring Album Art" (page 64). The time offset from track start is the value returned by "Command: 0x002A: GetTrackArtworkTimes" (page 430).

Table 5-23 `GetTrackArtworkData` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	trackIndex .
0x*NN*	2	formatID
0x*NN*	4	time offset from track start, in ms
Packet footer specified in "Command Packets" (page 109).		

5.1.18 Command 0x0011: RetTrackArtworkData

Lingo: 0x04 — Origin: Apple device

Applies To: Playback Engine

The Apple device sends the requested artwork to the accessory. Multiple `RetTrackArtworkData` commands may be necessary to transfer all the data because it will be too much to fit into a single packet. See "Transferring Album Art" (page 64).

This command uses nearly the same format as the `SetDisplayImage` command (command 0x0032). The only difference is the addition of 2 coordinates; they define an inset rectangle that describes any padding that may have been added to the image. The coordinates consist of two x,y pairs. Each x or y value is 2 bytes, so the total size of the coordinate set is 8 bytes.

Table 5-24 `RetTrackArtworkData` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		

Value	Bytes	Parameter
0x00	2	Descriptor packet index. This field uniquely identify each packet in the `RetTrackArtworkData` transaction. The first command is the descriptor packet, which always starts with an index of 0x0000.
0x*NN*	1	Display pixel format code.
0x*NN*	2	Image width in pixels
0x*NN*	2	Image height in pixels
0x*NN*	2	Inset rectangle, top-left point, x value
0x*NN*	2	Inset rectangle, top-left point, y value
0x*NN*	2	Inset rectangle, bottom-right point, x value
0x*NN*	2	Inset rectangle, bottom-right point, y value
0x*NN*	4	Row size in bytes
0x*NN*	*NN*	Image pixel data (variable length)
Packet footer specified in "Command Packets" (page 109).		

In subsequent packets in the sequence, fields "Display pixel format code" through "Row size in bytes" are omitted. The only parameters sent are the Descriptor packet index (greater than 0x0000) and the Image pixel data.

5.1.19 Command 0x0016: ResetDBSelection

Lingo: 0x04 — Origin: Accessory

Applies To: Database Engine

Resets the current database selection to an empty state, invalidates the category entry count (sets the count to 0) for all categories except the playlist category, and sets the database hierarchy to the audio hierarchy (even if it is currently in the video hierarchy). This is analogous to pressing a clickwheel Apple device's Menu button repeatedly to get to its top-level display. Any previously selected database items are deselected. The command has no effect on the Playback Engine. In response, the Apple device sends an `iPodAck` command with the command status.

Once the accessory has reset the database selection, it must initialize the category count before it can select database records. Please refer to "Command 0x0018: GetNumberCategorizedDBRecords" (page 417) and "Command 0x0017: SelectDBRecord" (page 415) for details.

Note: Starting with protocol version 1.07, the `ResetDBSelection` command clears the sort order.

Table 5-25 `ResetDBSelection` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

5.1.20 Command 0x0017: SelectDBRecord

Lingo: 0x04 — Origin: Accessory

Applies To: Database Engine. Selecting a single track automatically passes it to the Playback Engine.

Selects one or more records in the Database Engine, based on a category relative index. For example, selecting category two (artist) and record index one results in a list of selected tracks (or database records) from the second artist in the artist list. Table 5-27 (page 416) lists the available database categories.

Note: With iOS devices, selecting a specific track in a playlist containing only skip-when-shuffle tracks causes the entire playlist to be shuffled and passed to the Playback Engine. With other Apple devices, passing a valid track index with `SelectDBRecord` and a playlist containing only skip-when-shuffle tracks causes a single track to be passed to the Playback Engine. Use the `GetNumPlayingTracks` command to query the number of songs in the Playback Engine before using any other Playback Engine commands to alter playback.

Selections are additive and limited by the category hierarchy; see "Database Category Hierarchies" (page 60) for more information about category hierarchies. Subsequent selections are made based on the subset of records resulting from the previous selections and not from the entire database. Note that the selection of a single record automatically passes it to the Playback Engine and starts its playback. Record indices consist of a 32-bit signed integer.

`SelectDBRecord` may be called only after a category count has been initialized through a call to "Command 0x0018: GetNumberCategorizedDBRecords" (page 417). Without a valid category count, the `SelectDBRecord` call cannot select a database record and the result of calling it will be undefined. Accessories that make use of "Command 0x0016: ResetDBSelection" (page 414) must always initialize the category count before selecting a new database record using `SelectDBRecord`.

Accessories should pay close attention to the `iPodAck` returned by the `SelectDBRecord` command. Ignoring errors may cause unexpected behavior.

To undo a database selection, send the `SelectDBRecord` command with the current category selected in the Database Engine and a record index of –1 (0xFFFFFFFF). This has the same effect as pressing the iPod Menu button once and moves the database selection up to the next highest menu level. For example, if an accessory selected artist number three and then album number one, it could use the `SelectDBRecord(Album, -1)` command to return to the database selection of artist number three. If

multiple database selections have been made, accessories can use any of the previously used categories to return to the next highest database selection. If the category used in one of these `SelectDBRecord` commands has not been used in a previous database selection then the command is treated as a no-op.

Sending a `SelectDBRecord` command with the Track or Audiobook category and a record index of –1 is invalid, because the previous database selection made with the Track category and a valid index passes the database selection to the Playback Engine. Sending a `SelectDBRecord(Track, -1)` command returns a parameter error. The Apple device also returns a bad parameter error `iPodAck` when accessories send the `SelectDBRecord` command with an invalid category type, or with the Track category and an index greater than the total number of tracks available on the Apple device.

Note: Selecting a podcast always selects from the main podcast library regardless of the current category context of the Apple device. Similarly, selecting an audiobook track selects from the list of all audiobook tracks.

To immediately go to the topmost iPod menu level and reset all database selections, send the `ResetDBSelection` command to the Apple device.

Table 5-26 `SelectDBRecord` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Database category type. See Table 5-27 (page 416).
0x*NN*	4	Database record index
Packet footer specified in "Command Packets" (page 109).		

Table 5-27 (page 416) lists the valid database categories.

Table 5-27 Database category types for commands

Category	Code	Protocol version	Comments
Top-level	0x00	1.14	`ReturnCategorizedDatabaseRecord` returns a list of top-level category names and index values. If the Apple device does not support a category, the returned name will be a null string; that category must not be used by `SelectDBRecord` commands.
Playlist	0x01	1.00	
Artist	0x02	1.00	
Album	0x03	1.00	
Genre	0x04	1.00	
Track	0x05	1.00	
Composer	0x06	1.00	

Category	Code	Protocol version	Comments
Audiobook	0x07	1.06	
Podcast	0x08	1.08	
Nested playlist	0x09	1.13	
Genius Mixes	0x0A	1.14	Similar to playlists, but with the Genius feature.
iTunesU	0x0B	1.14	Similar to podcasts
Reserved	0x0C – 0xFF	N/A	

5.1.21 Command 0x0018: GetNumberCategorizedDBRecords

Lingo: 0x04 — Origin: Accessory

Applies To: Database Engine

Retrieves the number of records in a particular database category. For example, an accessory can get the number of artists or albums present in the database. The category types are described in Table 5-27 (page 416). The Apple device responds with a "Command 0x0019: ReturnNumberCategorizedDBRecords" (page 418) command indicating the number of records present for this category.

`GetNumberCategorizedDBRecords` must be called to initialize the category count before selecting a database record using "Command 0x0017: SelectDBRecord" (page 415). A category's record count can change based on the prior categories selected and the database hierarchy. The accessory is expected to call `GetNumberCategorizedDBRecords` in order to get the valid range of category entries before selecting a record in that category.

Note: The record count returned by this command depends on the database state before this command is sent. If the database has been reset using "Command 0x0016: ResetDBSelection" (page 414), this command returns the total number of records for a given category. However, if this command is sent after one or more categories are selected, the record count is the subset of records that are members of all the categories selected prior to this command.

Table 5-28 `GetNumberCategorizedDBRecords` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Database category type. See Table 5-27 (page 416).
Packet footer specified in "Command Packets" (page 109).		

5.1.22 Command 0x0019: ReturnNumberCategorizedDBRecords

Lingo: 0x04 — Origin: Apple device

Returns the number of database records matching the specified database category. The Apple device sends this command in response to the "Command 0x0018: GetNumberCategorizedDBRecords" (page 417) command from the accessory. Individual records can then be extracted by sending "Command 0x001A: RetrieveCategorizedDatabaseRecords" (page 418) to the Apple device.

If no matching database records are found, a record count of zero is returned. Category types are described in Table 5-27 (page 416).

After selecting the podcast category, the number of artist, album, composer, genre, and audiobook records is always zero.

Table 5-29 `ReturnNumberCategorizedDBRecords` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Database record count
Packet footer specified in "Command Packets" (page 109).		

5.1.23 Command 0x001A: RetrieveCategorizedDatabaseRecords

Lingo: 0x04 — Origin: Accessory

Applies To: Database Engine

Retrieves one or more database records from the Apple device, typically based on the results from the "Command 0x0018: GetNumberCategorizedDBRecords" (page 417) query. The database category types are described in Table 5-27 (page 416).

This command specifies the starting record index and the number of records to retrieve (the record count). This allows an accessory to retrieve an individual record or the entire set of records for a category. The record start index and record count consist of 32-bit signed integers. To retrieve all records from a given starting record index, set the record count to –1 (0xFFFFFFFF).

The Apple device responds to this command with a separate "Command 0x001B: ReturnCategorizedDatabaseRecord" (page 419) command for each record matching the specified criteria (category and record index range).

Table 5-30 `RetrieveCategorizedDatabaseRecords` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Database category type. See Table 5-27 (page 416).

Value	Bytes	Parameter
0x*NN*	4	Database record start index
0x*NN*	4	Database record read count
Packet footer specified in "Command Packets" (page 109).		

5.1.24 Command 0x001B: ReturnCategorizedDatabaseRecord

Lingo: 0x04 — Origin: Apple device

Contains information for a single database record. The Apple device sends one or more of these commands in response to the "Command 0x001A: RetrieveCategorizedDatabaseRecords" (page 418) command from the accessory. The category record index is included to allow the accessory to determine which record has been sent. The record data is sent as a null-terminated UTF-8 encoded data array.

Note: The database record string is not limited to 252 characters; it may be sent in small or large packet format, depending on the record size. The small packet format is shown.

Table 5-31 `ReturnCategorizedDatabaseRecord` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Database record category index
0x*NN*	*NN*	Database record as a UTF-8 character array.
Packet footer specified in "Command Packets" (page 109).		

5.1.25 Command 0x001C: GetPlayStatus

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

Requests the current Apple device playback status, allowing the accessory to display feedback to the user. In response, the Apple device sends a "Command 0x001D: ReturnPlayStatus" (page 420) command with the current playback status.

Table 5-32 `GetPlayStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		

Value	Bytes	Parameter
Packet footer specified in "Command Packets" (page 109).		

5.1.26 Command 0x001D: ReturnPlayStatus

Lingo: 0x04 — Origin: Apple device

Returns the current Apple device playback status. The Apple device sends this command in response to the "Command 0x001C: GetPlayStatus" (page 419) command from the accessory. The information returned includes the current track length, track position, and player state.

Note: The track length and track position fields are valid only if the player state is Playing or Paused. For other player states, these fields must be ignored.

Table 5-33 `ReturnPlayStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Track length in milliseconds
0x*NN*	4	Track position in milliseconds
0x*NN*	1	Player state. Possible values are: ■ 0x00 = Stopped ■ 0x01 = Playing ■ 0x02 = Paused ■ 0x03 – 0xFE = Reserved ■ 0xFF = Error
Packet footer specified in "Command Packets" (page 109).		

5.1.27 Command 0x001E: GetCurrentPlayingTrackIndex

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

Requests the Playback Engine index of the currently playing track. In response, the Apple device sends a "Command 0x001F: ReturnCurrentPlayingTrackIndex" (page 421) command to the accessory.

Note: The track index returned is valid only if there is currently a track playing or paused.

Table 5-34 `GetCurrentPlayingTrackIndex` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

5.1.28 Command 0x001F: ReturnCurrentPlayingTrackIndex

Lingo: 0x04 — Origin: Apple device

Returns the Playback Engine index of the current playing track in response to the "Command 0x001E: GetCurrentPlayingTrackIndex" (page 420) command from the accessory. The track index is a 32-bit signed integer. If there is no track currently playing or paused, an index of –1 (0xFFFFFFFF) is returned.

Table 5-35 `ReturnCurrentPlayingTrackIndex` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Playback track index
Packet footer specified in "Command Packets" (page 109).		

5.1.29 Command 0x0020: GetIndexedPlayingTrackTitle

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

Requests the title name of the indexed playing track from the Apple device. In response to a valid command, the Apple device sends a "Command 0x0021: ReturnIndexedPlayingTrackTitle" (page 422) command to the accessory.

Note: If the packet length or playing track index is invalid, the Apple device responds with an `iPodAck` command including the specific error status.

Table 5-36 `GetIndexedPlayingTrackTitle` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		

Value	Bytes	Parameter
0x*NN*	4	Playback track index
Packet footer specified in "Command Packets" (page 109).		

5.1.30 Command 0x0021: ReturnIndexedPlayingTrackTitle

Lingo: 0x04 — Origin: Apple device

Returns the title of the indexed playing track in response to a valid "Command 0x0020: GetIndexedPlayingTrackTitle" (page 421) command from the accessory. The track title is encoded as a null-terminated UTF-8 character array.

Note: The track title string is not limited to 252 characters; it may be sent in small or large packet format, depending on the string length. The small packet format is shown.

Table 5-37 `ReturnIndexedPlayingTrackTitle` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	*NN*	Track title as a UTF-8 character array. The Apple device may send this data in any format defined in "Command Packets" (page 109). Accessories must be prepared to receive packets of any size unless they have specified a maximum packet payload length through a `RetAccessoryInfo` command.
Packet footer specified in "Command Packets" (page 109).		

5.1.31 Command 0x0022: GetIndexedPlayingTrackArtistName

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

Requests the name of the artist of the indexed playing track. In response to a valid command, the Apple device sends a "Command 0x0023: ReturnIndexedPlayingTrackArtistName" (page 423) command to the accessory.

Note: If the packet length or playing track index is invalid, the Apple device responds with an `iPodAck` command including the specific error status.

Table 5-38 `GetIndexedPlayingTrackArtistName` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Playback track index
Packet footer specified in "Command Packets" (page 109).		

5.1.32 Command 0x0023: ReturnIndexedPlayingTrackArtistName

Lingo: 0x04 — Origin: Apple device

Returns the artist name of the indexed playing track in response to a valid "Command 0x0022: GetIndexedPlayingTrackArtistName" (page 422) command from the accessory. The track artist name is encoded as a null-terminated UTF-8 character array.

Note: The artist name string is not limited to 252 characters; it may be sent in small or large packet format, depending on the string length. The small packet format is shown.

Table 5-39 `ReturnIndexedPlayingTrackArtistName` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	*NN*	Artist name as UTF-8 character array
Packet footer specified in "Command Packets" (page 109).		

5.1.33 Command 0x0024: GetIndexedPlayingTrackAlbumName

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

Requests the album name of the indexed playing track. In response to a valid command, the Apple device sends a "Command 0x0025: ReturnIndexedPlayingTrackAlbumName" (page 424) command to the accessory.

Note: If the received packet length or playing track index is invalid, the Apple device responds with an `iPodAck` command including the specific error status.

Table 5-40 `GetIndexedPlayingTrackAlbumName` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Playback track index
Packet footer specified in "Command Packets" (page 109).		

5.1.34 Command 0x0025: ReturnIndexedPlayingTrackAlbumName

Lingo: 0x04 — Origin: Apple device

Returns the album name of the indexed playing track in response to a valid "Command 0x0024: GetIndexedPlayingTrackAlbumName" (page 423) command from the accessory. The track album name is encoded as a null-terminated UTF-8 character array.

Note: The album name string is not limited to 252 characters.

Table 5-41 `ReturnIndexedPlayingTrackAlbumName` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	*NN*	Album name as a UTF-8 character array
Packet footer specified in "Command Packets" (page 109).		

5.1.35 Command 0x0026: SetPlayStatusChangeNotification

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

This command is sent by the accessory to control the status change event types sent by the Apple device.

There are two forms of the command. The one-byte form accepts a single Boolean to enable or disable all notifications for play state, track index, track time position, FFW/REW seek stop, and chapter index changes (see Table 5-47 (page 426), "Play status change notification codes"). The packet for this form of the command is shown in Table 5-42 (page 425). When the value of its parameter is set to 0x00, all notifications are disabled, whether previously set by the one-byte or four-byte form of the command.

The four-byte form expands the status notification control into individual bits for each type of status. The packet for this form of the command is shown in Table 5-43 (page 425). Its parameter is a fixed-length bitmask of events to be enabled or disabled, listed in Table 5-45 (page 425), where 1 = enable, 0 = disable.

Note: The accessory must send the four-byte form first; if the `iPodAck` command that the Apple device returns passes a status other than 0x00 (Success), then the accessory may send the one-byte form.

Table 5-42 One-byte `SetPlayStatusChangeNotification` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Enable/disable notifications; see Table 5-44 (page 425).
Packet footer specified in "Command Packets" (page 109).		

Table 5-43 Four-byte `SetPlayStatusChangeNotification` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x00	4	Notification event mask; see Table 5-45 (page 425).
Packet footer specified in "Command Packets" (page 109).		

Table 5-44 One-byte status change event values

Value	Description
0x00	Disable all status event notifications
0x01	Enable play status notifications for basic play state, track index, track time position, FFW/REW seek stop, and chapter index changes (see Table 5-47 (page 426), "Play status change notification codes").
0x02-0xFF	Reserved

Table 5-45 Four-byte status change event mask bits

Bit	Description
00	Basic play state changes (stop, FFW seek stop, or REW seek stop, using status notification types 0x00, 0x02, or 0x03).
01	Extended play state changes (playback stop, FFW seek start, REW seek start, playback started, FFW/REW seek stop, or playback pause using status notification type 0x06). Uses `PlayControl` command control codes as the play status codes; see Table 5-49 (page 429).
02	Track index

Bit	Description
03	Track time offset (ms)
04	Track time offset (sec)
05	Chapter index
06	Chapter time offset (ms)
07	Chapter time offset (sec)
08	Track unique identifier
09	Track media type (audio/video)
10	Track lyrics ready (if the track has lyrics)
11	Track capabilities changed
12	Playback engine contents changed
31:13	Reserved

5.1.36 Command 0x0027: PlayStatusChangeNotification

Lingo: 0x04 — Origin: Apple device

This command is sent by the Apple device to notify the accessory about Extended Interface mode status changes. The types of status notifications sent to the accessory are controlled by the "Command 0x0026: SetPlayStatusChangeNotification" (page 424) command.

Table 5-46 `PlayStatusChangeNotification` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	*NN*	New play status. See Table 5-47 (page 426).
Packet footer specified in "Command Packets" (page 109).		

Table 5-47 Play status change notification codes

Status change	Parameters:bytes	Description
Playback stopped	`{0x00}`	
Track index	`{0x01, trackIndex:4}`	`trackIndex` = playback engine track index
Playback FFW seek stop	`{0x02}`	

Status change	Parameters:bytes	Description
Playback REW seek stop	{0x03}	
Track time offset	{0x04, trackOffsetMs:4}	trackOffsetMs = track time offset in milliseconds
Chapter index	{0x05, chapIndex:4}	chapIndex = chapter index
Playback status extended	{0x06, playState:1}	playState = playback state: 0x00-0x01 = Reserved 0x02 = Stopped 0x03-0x04 = Reserved 0x05 = FFW seek started 0x06 = REW seek started 0x07 = FFW/REW seek stopped 0x08-0x09 = Reserved 0x0A = Playing 0x0B = Paused 0x0C-0xFF = Reserved
Track time offset	{0x07, trackOffsetSec:4}	trackOffsetSec = track time offset in seconds
Chapter time offset (ms)	{0x08, chapTimeMs:4}	chapTimeMs = chapter time offset in milliseconds
Chapter time offset (sec)	{0x09, chapTimeSec:4}	chapTimeSec = chapter time offset in seconds
Track unique identifier	{0x0A, trackUID:8}	trackUID = track unique identifier
Track playback mode	{0x0B, playMode:1}	playMode = playback mode of current playing track (see Note, below): 0x00 = Audio track 0x01 = Video track 0x02-0xFF = Reserved
Track lyrics ready	{0x0C}	Lyrics for the currently playing track are available for download

Status change	Parameters:bytes	Description
Track capabilities changed	{0x0D, trackCapabilities:4}	Track capabilities bits: Bit 31:15: Reserved Bit 14: Track is an iTunesU episode Bit 13: Track is capable of generating a Genius playlist Bit 12:9: Reserved Bit 8: Track is currently queued to play as a video Bit 7: Track contains video (a video podcast, music video, movie, or TV show) Bit 6: Track has description Bit 5: Track has release date Bit 4: Track is a podcast episode Bit 3: Track has song lyrics Bit 2: Track has album artwork Bit 1: Track has chapters Bit 0: Track is audiobook
Playback engine contents changed	{0x0E, numTracks:4}	Number of tracks in the new playlist
Reserved	{0x0F-0xFF}	

Note: The playback mode when a track is playing may depend on the database hierarchy that was current when the track was selected. Hybrid video/audio tracks (such as music videos or video podcasts) are queued as audio tracks if selected in the audio DB hierarchy, or queued as video tracks if selected in the video DB hierarchy.

5.1.37 Command 0x0029: PlayControl

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

This command is sent by the accessory to control the media playback state of the Apple device. If the Apple device is already in the requested state, the command has no effect and returns successfully. If the Apple device does not enter the requested state successfully, an error status is returned. In response, the Apple device sends an `iPodAck` command with the play control status.

Note: The iPod models before the 2G nano (09/2006) always return a successful status. Starting with the 2G nano, Apple devices return the actual play control status. This means that a next or previous track command will return an error if no media is playing.

Table 5-48 `PlayControl` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Play control command code. See Table 5-49 (page 429).
Packet footer specified in "Command Packets" (page 109).		

Table 5-49 (page 429) shows the Apple device play states the accessory can set.

Table 5-49 Play control command codes

Command	Code	Protocol	Comments
Reserved	0x00	Reserved	
Toggle Play/Pause	0x01	1.00	
Stop	0x02	1.00	
Next Track	0x03	1.00	**Deprecated; use Next (0x08) and Previous (0x09) instead.**
Previous Track	0x04	1.00	
Start FF	0x05	1.00	These commands must be followed by an End FF/Rew command (code 0x07) before any other `PlayControl` commands may be sent.
Start Rew	0x06	1.00	
End FF/Rew	0x07	1.00	
Next	0x08	1.06	If a track has chapters, Next (0x08) will advance to the beginning of the next chapter. If the track is playing its last chapter or has no chapters, Next will advance to the beginning of the next track. If a track has chapters and is more than two seconds into the current chapter, Previous (0x09) will back up to the beginning of the current chapter. If it is less than two seconds into the current chapter, Previous will back up to the beginning of the previous chapter. If the track has no chapters, Previous will back up to the beginning of the previous track.
Previous	0x09	1.06	
Play	0x0A	1.13	These commands may be used regardless of the lingo protocol version if the Apple device explicitly declares their support; see Table 3-132 (page 192).
Pause	0x0B	1.13	
Next Chapter	0x0C	1.14	**Deprecated; use Next (0x08) and Previous (0x09) instead.**

Command	Code	Protocol	Comments
Previous Chapter	0x0D	1.14	
Resume iPod	0x0E	1.14	Resume playback using the built-in Music app, which becomes the NowPlaying app. Use of this command is restricted; see the warning at the end of "Interaction with iOS Media Applications" (page 54).
Reserved	0x0F – 0xFF	N/A	

5.1.38 Command: 0x002A: GetTrackArtworkTimes

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

The accessory sends this command to the Apple device to request the list of artwork time locations for a track. A 4-byte `trackIndex` specifies which track from the Playback Engine is to be selected. A 2-byte `formatID` indicates which type of artwork is desired. See "Transferring Album Art" (page 64).

The 2-byte `artworkIndex` specifies at which index to begin searching for artwork. A value of 0 indicates that the Apple device should start with the first available artwork.

The 2-byte `artworkCount` specifies the maximum number of times (artwork locations) to be returned. A value of –1 (0xFFFF) indicates that there is no preferred limit. Note that podcasts may have a large number of associated images.

Table 5-50 `GetTrackArtworkTimes` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	trackIndex
0x*NN*	2	formatID
0x*NN*	2	artworkIndex
0x*NN*	2	artworkCount
Packet footer specified in "Command Packets" (page 109).		

5.1.39 Command: 0x002B: RetTrackArtworkTimes

Lingo: 0x04 — Origin: Apple device

Applies To: Playback Engine

The Apple device sends this command to the accessory to return the list of artwork times for a given track. The Apple device returns zero or more 4-byte times, one for each piece of artwork associated with the track and format specified by `GetTrackArtworkTimes`. See "Transferring Album Art" (page 64).

The number of records returned will be no greater than the number specified in the `GetTrackArtworkTimes` command. It may, however, be less than requested. This can happen if there are fewer pieces of artwork available than were requested, or if the Apple device is unable to place the full number in a single packet. Check the number of records returned against the results of `ReturnIndexedPlayingTrackInfo` with `infoType` 8 to ensure that all artwork has been received.

Table 5-51 `RetTrackArtworkTimes` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	time offset from track start in ms
0x*NN*	*NN*	Preceding field may be repeated *NN* times
Packet footer specified in "Command Packets" (page 109).		

5.1.40 Command 0x002C: GetShuffle

Lingo: 0x04 — Origin: Accessory

Requests the current state of the Apple device shuffle setting. The Apple device responds with the "Command 0x002D: ReturnShuffle" (page 431) command.

Table 5-52 `GetShuffle` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

5.1.41 Command 0x002D: ReturnShuffle

Lingo: 0x04 — Origin: Apple device

Returns the current state of the shuffle setting. The Apple device sends this command in response to the "Command 0x002C: GetShuffle" (page 431) command from the accessory.

Table 5-53 `ReturnShuffle` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		

Value	Bytes	Parameter
0x*NN*	1	Shuffle mode. See Table 5-54 (page 432).
Packet footer specified in "Command Packets" (page 109).		

Table 5-54 (page 432) lists the possible values of the shuffle mode.

Table 5-54 Shuffle modes

Value	Meaning
0x00	Shuffle off
0x01	Shuffle tracks
0x02	Shuffle albums
0x03 – 0xFF	Reserved

5.1.42 Command 0x002E: SetShuffle

Lingo: 0x04 — Origin: Accessory

Sets the Apple device shuffle mode. The Apple device shuffle modes are listed in Table 5-54 (page 432). In response, the Apple device sends an `iPodAck` command with the command status.

This command has an optional field called Restore on Exit. This field can be used to restore the original shuffle setting in use when the accessory was attached to the Apple device. A nonzero value restores the original shuffle setting of the Apple device when the accessory is detached. If this field is zero, the shuffle setting set by the accessory overwrites the original setting and persists after the accessory is detached from the Apple device.

Accessory engineers should note that the shuffle mode affects items only in the Playback Engine. The shuffle setting does not affect the order of tracks in the database engine, so calling "Command 0x001A: RetrieveCategorizedDatabaseRecords" (page 418) on a database selection with the shuffle mode set returns a list of unshuffled tracks. To get the shuffled playlist, an accessory must query the Playback Engine by calling "Command 0x0020: GetIndexedPlayingTrackTitle" (page 421).

Shuffling tracks does not affect the track index, just the track at that index. If an unshuffled track at playback index 1 is shuffled, a new track is placed into index 1. The playback indexes themselves are not shuffled.

When shuffle mode is enabled, tracks that are marked "skip when shuffling" are filtered from the database selection. This affects all tracks that the user has marked in iTunes. It also affects all podcasts unless their default "skip when shuffling" markings have been deliberately removed. To apply the filter to the Playback Engine, the accessory must send "Command 0x0017: SelectDBRecord" (page 415) after enabling shuffle mode.

Note: Accessory developers are encouraged to always use the Restore on Exit field with a nonzero value to restore any settings modified by the accessory upon detach.

Table 5-55 (page 433) shows a command to set the shuffle setting and optionally restore it on accessory detach.

Table 5-55 `SetShuffle` packet with Restore on Exit field

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	New shuffle mode. See Table 5-54 (page 432).
0x*NN*	1	Restore on Exit: If 1, the original setting is restored on detach; if 0, the new setting persists after accessory detach.
Packet footer specified in "Command Packets" (page 109).		

Table 5-56 (page 433) shows a command to make the shuffle setting persistent after the accessory detach.

Table 5-56 `SetShuffle` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	New shuffle mode. See Table 5-54 (page 432).
Packet footer specified in "Command Packets" (page 109).		

5.1.43 Command 0x002F: GetRepeat

Lingo: 0x04 — Origin: Accessory

Requests the track repeat state of the Apple device. In response, the Apple device sends a "Command 0x0030: ReturnRepeat" (page 434) command to the accessory.

Table 5-57 `GetRepeat` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

5.1.44 Command 0x0030: ReturnRepeat

Lingo: 0x04 — Origin:

Returns the current Apple device track repeat state to the accessory. The Apple device sends this command in response to the "Command 0x002F: GetRepeat" (page 433) command.

Table 5-58 `ReturnRepeat` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Repeat state. See Table 5-59 (page 434).
Packet footer specified in "Command Packets" (page 109).		

Table 5-59 (page 434) lists the possible values of the repeat state field.

Table 5-59 Repeat state values

Value	Meaning
0x00	Repeat off
0x01	Repeat one track
0x02	Repeat all tracks
0x03 – 0xFF	Reserved

5.1.45 Command 0x0031: SetRepeat

Lingo: 0x04 — Origin: Accessory

Sets the repeat state of the Apple device. The Apple device track repeat modes are listed in Table 5-59 (page 434). In response, the Apple device sends an `iPodAck` command with the command status.

This command has an optional field called Restore on Exit. This field can be used to restore the original repeat setting in use when the accessory was attached to the Apple device. A nonzero value restores the original repeat setting of the Apple device when the accessory is detached. If this field is zero, the repeat setting set by the accessory overwrites the original setting and persists after the accessory is detached from the Apple device.

Note: Accessory developers are encouraged to always use the Restore on Exit field with a nonzero value to restore any settings modified by the accessory upon detach.

Table 5-60 (page 435) shows a command to set the repeat setting and optionally restore it on accessory detach.

Table 5-60 `SetRepeat` packet with Restore on Exit field

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	New repeat state. See Table 5-59 (page 434).
0x*NN*	1	Restore on Exit: If 1, the original setting is restored on detach; if 0, the new setting persists after accessory detach.
Packet footer specified in "Command Packets" (page 109).		

Table 5-61 (page 435) shows a command to make the repeat setting persistent after the accessory detach.

Table 5-61 `SetRepeat` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	New repeat state. See Table 5-59 (page 434).
Packet footer specified in "Command Packets" (page 109).		

5.1.46 Command 0x0032: SetDisplayImage

Lingo: 0x04 — Origin: Accessory

IMPORTANT: This technology is deprecated. Support for it is not guanteed in future Apple devices and it must not be used in new accessory designs.

Sets a bitmap image that is shown on the Apple device display when it is connected to the accessory. The intent is to allow third party branding when the Apple device is communicating with an external accessory. An image downloaded using this mechanism replaces the default checkmark bitmap image that is displayed when the Apple device is connected to an external accessory. The new bitmap is retained in RAM for as long as the Apple device remains powered. After a system reset or Sleep state, the new bitmap is lost and the checkmark image restored as the default when the Apple device next enters Extended Interface mode after power-up.

Note: To determine whether the Apple device supports displaying the image sent by `SetDisplayImage`, the accessory should verify that General lingo command `RetiPodOptionsForLingo` returns bit 04 for lingo 0x04, as listed in Table 3-132 (page 192).

Before setting a monochrome display image, the accessory can send the "Command 0x0033: GetMonoDisplayImageLimits" (page 440) command to obtain the current Apple device display width, height and pixel format. The monochrome display information returned in the "Command 0x0034: ReturnMonoDisplayImageLimits" (page 440) command can be useful to the accessory in deciding which type of display image format is suitable for downloading to the Apple device.

On Apple devices with color displays, accessories can send the "Command 0x0039: GetColorDisplayImageLimits" (page 442) command to obtain the Apple device color display width, height, and pixel formats. The color display information is returned in the "Command 0x003A: ReturnColorDisplayImageLimits" (page 443) command.

To set a display image, the accessory must successfully send `SetDisplayImage` descriptor and data commands to the Apple device. The `SetDisplayImage` descriptor command (packet index 0x0000) must be sent first, as it gives the Apple device a description of the image to be downloaded. This command is shown in Table 5-62 (page 436)). The image descriptor command includes image pixel format, image width and height, and display row size (stride) in bytes. Optionally, the descriptor command may also contain the beginning data of the display image, as long as it remains within the maximum length limits of the packet.

Following the descriptor command, the `SetDisplayImage` data commands (packet index 0x0001 – 0x*NNNN*) must be sent using sequential packet indices until the entire image has been sent to the Apple device.

Note: The `SetDisplayImage` command payload length is limited to the maximum value determined by calling `RequestTransportMaxPayloadSize`. This packet length limit and the size and format of the display image generally determine the minimum number of packets that are required to set a display image.

Note: Starting with the second generation iPod nano with version 1.1.2 firmware, using the `SetDisplayImage` command is limited to once every 15 seconds over USB transport. The iPod classic and iPod 3G nano apply a different restriction to both USB and UART transports: calls made to `SetDisplayImage` after the first 15 seconds will return a successful `iPodAck` command, but the bitmap will not be displayed on the Apple device's screen. Hence, use of the `SetDisplayImage` command should be limited to drawing bitmap images only immediately after entering Extended mode. The iPod touch accepts the `SetDisplayImage` command but will not draw it on its screen.

Table 5-62 (page 436) shows the format of a descriptor command. This example assumes the display image descriptor data exceeds the small packet payload capacity; a large packet format is shown.

Table 5-62 `SetDisplayImage` descriptor command (packet index = 0x0000)

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x00	2	Descriptor command index. This field uniquely identifies each packet in the `SetDisplayImage` transaction. The first command is the Descriptor command and always starts with an index of 0x0000.
0x*NN*	1	Display pixel format code. See Table 5-64 (page 438).
0x*NN*	2	Image width in pixels. The number of pixels, from left to right, per row.
0x*NN*	2	Image height in pixels. The number of rows, from top to bottom, in the image.
0x*NN*	4	Row size (stride) in bytes. The number of bytes representing one row of pixels. Each row is zero-padded to end on a 32-bit boundary. The cumulative size, in bytes, of the image data, transferred across all packets in this transaction is effectively `(Row Size * Image Height)`.
0x*NN*	1	Display image pixel data

Value	Bytes	Parameter
Packet footer specified in "Command Packets" (page 109).		

Note: Table 5-63 (page 437) shows the format of a data command. This example assumes the display image data exceeds the small packet payload capacity; a large packet format is shown.

Table 5-63 `SetDisplayImage` data packet (packet index = 0x0001 – 0xNNNN)

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	2	Descriptor command index. This field uniquely identifies each packet in the `SetDisplayImage` transaction. The first command is the descriptor command, shown in Table 5-62 (page 436). The remaining `n-1` commands are simply data packets, where `n` is determined by the size of the image.
0x*NN*	1	Display image pixel data
Packet footer specified in "Command Packets" (page 109).		

Note: A known issue causes `SetDisplayImage` data packet lengths less than 11 bytes to return a bad parameter error (0x04) `iPodAck` on 3G iPods.

The Apple device display is oriented as a rectangular grid of pixels. In the horizontal direction (x-coordinate), the pixel columns are numbered, left to right, from 0 to Cmax. In the vertical direction (y-coordinate), the pixel rows are numbered, top to bottom, from 0 to Rmax. Therefore, an (x,y) coordinate of (0,0) represents the upper-leftmost pixel on the display and (Cmax,Rmax) represents the lower-rightmost pixel on the display. A portion of the Apple device display pixel layout is shown below, where x is the column number, y is the row number, and (Cmax,Rmax) represents the maximum row and column numbers.

Pixel layout	*x*									
y	0,0	1,0	2,0	3,0	4,0	5,0	6,0	7,0	…	Cmax,0
	0,1	1,1	2,1	3,1	4,1	5,1	6,1	7,1	…	Cmax,1
	0,2	1,2	2,2	3,2	4,2	5,2	6,2	7,2	…	Cmax,2
	0,3	1,3	2,3	3,3	4,3	5,3	6,3	7,3	…	Cmax,3
	0,4	1,4	2,4	3,4	4,4	5,4	6,4	7,4	…	Cmax,4
	0,5	1,5	2,5	3,5	4,5	5,5	6,5	7,5	…	Cmax,5
	0,6	1,6	2,6	3,6	4,6	5,6	6,6	7,6	…	Cmax,6
	0,7	1,7	2,7	3,7	4,7	5,7	6,7	7,7	…	Cmax,7

	…	…	…	…	…	…	…	…	…	…
	0, Rmax	1, Rmax	2, Rmax	3, Rmax	4, Rmax	5, Rmax	6, Rmax	7, Rmax	…	Cmax,Rmax

The Apple device display pixel formats are shown in Table 5-64 (page 438).

Table 5-64 Display pixel format codes

Display pixel format	Code	Protocol version
Reserved	0x00	N/A
Monochrome, 2 bits per pixel	0x01	1.01
RGB 565 color, little-endian, 16 bpp	0x02	1.09
RGB 565 color, big-endian, 16 bpp	0x03	1.09
Reserved	0x04 – 0xFF	N/A

Apple devices with color screens support all three image formats. All other Apple devices support only display pixel format 0x01 (monochrome, 2 bpp).

5.1.46.1 Display Pixel Format 0x01

Display pixel format 0x01 (monochrome, 2 bits per pixel) is the pixel format supported by all Apple devices. Each pixel consists of 2 bits that control the pixel intensity. The pixel intensities and associated binary codes are listed in Table 5-65 (page 438).

Table 5-65 2 bpp monochrome pixel intensities

Pixel Intensity	Binary Code
Pixel off (not visible)	00b
Pixel on 25% (light grey)	01b
Pixel on 50% (dark grey)	10b
Pixel on 100% (black)	11b

Each byte of image data contains four packed pixels. The pixel ordering within bytes and the byte ordering within 32 bits is shown below.

Image Data Byte 0x0000							
Pixel 0		Pixel 1		Pixel 2		Pixel 3	
7	6	5	4	3	2	1	0

Image Data Byte 0x0001							
Pixel 4		Pixel 5		Pixel 6		Pixel 7	
7	6	5	4	3	2	1	0

Image Data Byte 0x0002							
Pixel 8		Pixel 9		Pixel 10		Pixel 11	
7	6	5	4	3	2	1	0

Image Data Byte 0x0003							
Pixel 12		Pixel 13		Pixel 14		Pixel 15	
7	6	5	4	3	2	1	0

Image Data Byte 0x*NNNN*							
Pixel N		Pixel N+1		Pixel N+2		Pixel N+3	
7	6	5	4	3	2	1	0

5.1.46.2 Display Pixel Formats 0x02 and 0x03

Display pixel format 0x02 (RGB 565, little-endian) and display pixel format 0x03 (RGB 565, big-endian) are available for use in all Apple devices with color screens. Each pixel consists of 16 bits that control the pixel intensity for the colors red, green, and blue.

It takes two bytes to represent a single pixel. Red is represented by 5 bits, green is represented by 6 bits, and blue by the final 5 bits. A 32-bit sequence represents 2 pixels. The pixel ordering within bytes and the byte ordering within 32 bits for display format 0x02 (RGB 565, little-endian) is shown below.

Image Data Byte 0x0000							
Pixel 0, lower 3 bits of green			Pixel 0, all 5 bits of blue				
7	6	5	4	3	2	1	0

Image Data Byte 0x0001							
Pixel 0, all 5 bits of red					Pixel 0, upper 3 bits of green		
7	6	5	4	3	2	1	0

Image Data Byte 0x0002							
Pixel 1, lower 3 bits of green			Pixel 1, all 5 bits of blue				
7	6	5	4	3	2	1	0

Image Data Byte 0x0003							
Pixel 1, all 5 bits of red					Pixel 1, upper 3 bits of green		
7	6	5	4	3	2	1	0

The format for display pixel format 0x03 (RGB 565, big-endian, 16 bpp) is almost identical, with the exception that bytes 0 and 1 are swapped and bytes 2 and 3 are swapped.

5.1.47 Command 0x0033: GetMonoDisplayImageLimits

Lingo: 0x04 — Origin: Accessory

Requests the limiting characteristics of the monochrome image that can be sent to the Apple device for display while it is connected to the accessory. It can be used to determine the display pixel format and maximum width and height of a monochrome image to be set using the "Command 0x0032: SetDisplayImage" (page 435) command. In response, the Apple device sends a "Command 0x0034: ReturnMonoDisplayImageLimits" (page 440) command to the accessory with the requested display information. The `GetMonoDisplayImageLimits` command is supported by Apple devices with either monochrome or color displays. To obtain color display image limits, use "Command 0x0039: GetColorDisplayImageLimits" (page 442).

Table 5-66 `GetMonoDisplayImageLimits` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

5.1.48 Command 0x0034: ReturnMonoDisplayImageLimits

Lingo: 0x04 — Origin: Apple device

Returns the limiting characteristics of the monochrome image that can be sent to the Apple device for display while it is connected to the accessory. The Apple device sends this command in response to the "Command 0x0033: GetMonoDisplayImageLimits" (page 440) command. Monochrome display characteristics include maximum image width and height and the display pixel format.

Table 5-67 `ReturnMonoDisplayImageLimits` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	2	Maximum image width in pixels
0x*NN*	2	Maximum image height in pixels
0x*NN*	1	Display pixel format (see Table 5-64 (page 438)).
Packet footer specified in "Command Packets" (page 109).		

5.1.49 Command 0x0035: GetNumPlayingTracks

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

Requests the number of tracks in the list of tracks queued to play on the Apple device. In response, the Apple device sends a "Command 0x0036: ReturnNumPlayingTracks" (page 441) command with the count of tracks queued to play.

Table 5-68 `GetNumPlayingTracks` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

5.1.50 Command 0x0036: ReturnNumPlayingTracks

Lingo: 0x04 — Origin: Apple device

Returns the number of tracks in the actual list of tracks queued to play, including the currently playing track (if any). The Apple device sends this command in response to the "Command 0x0035: GetNumPlayingTracks" (page 441) command.

Table 5-69 `ReturnNumPlayingTracks` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Number of tracks playing
Packet footer specified in "Command Packets" (page 109).		

5.1.51 Command 0x0037: SetCurrentPlayingTrack

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

Sets the index of the track to play in the Now Playing playlist on the Apple device. The index that is specified here is obtained by sending the "Command 0x0035: GetNumPlayingTracks" (page 441) and "Command 0x001E: GetCurrentPlayingTrackIndex" (page 420) commands to obtain the number of playing tracks and the current playing track index, respectively. In response, the Apple device sends an `iPodAck` command indicating the status of the command.

Note: This command is usable only when the Apple device is in a playing or paused state. If the Apple device is stopped, this command fails. If this command is sent with the current playing track index, the Apple device pauses playback momentarily and then resumes.

Table 5-70 `SetCurrentPlayingTrack` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	New current playing track index
Packet footer specified in "Command Packets" (page 109).		

5.1.52 Command 0x0039: GetColorDisplayImageLimits

Lingo: 0x04 — Origin: Accessory

Requests the limiting characteristics of the color image that can be sent to the Apple device for display while it is connected to the accessory. It can be used to determine the display pixel format and maximum width and height of a color image to be set using the "Command 0x0032: SetDisplayImage" (page 435) command. In response, the Apple device sends a "Command 0x003A: ReturnColorDisplayImageLimits" (page 443) command to the accessory with the requested display information. This command is supported only by Apple devices with color displays.

Table 5-71 `GetColorDisplayImageLimits` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

5.1.53 Command 0x003A: ReturnColorDisplayImageLimits

Lingo: 0x04 — Origin: Apple device

Returns the limiting characteristics of the color image that can be sent to the Apple device for display while it is connected to the accessory. The Apple device sends this command in response to the "Command 0x0039: GetColorDisplayImageLimits" (page 442) command. Display characteristics include maximum image width and height and the display pixel format.

Note: If the Apple device supports multiple display image formats, a 5-byte block of additional image width, height, and pixel format information is appended to the payload for each supported display format. The list of supported color display image formats returned by the Apple device may change in future software versions. Accessories must be able to parse a variable length list of supported color display formats to search for compatible formats.

Table 5-72 `ReturnColorDisplayImageLimits` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	2	Maximum image width in pixels
0x*NN*	2	Maximum image height in pixels
0x*NN*	1	Display pixel format (see Table 5-64 (page 438)).
0x*NN*	*NN*	Optional display image width, height, and pixel format for the second to nth supported display formats, if present.
Packet footer specified in "Command Packets" (page 109).		

5.1.54 Command 0x003B: ResetDBSelectionHierarchy

Lingo: 0x04 — Origin: Accessory

Applies To: Database Engine

This command passes a 1-byte parameter. A hierarchy selection value of 0x01 means that the accessory wants to navigate the audio hierarchy; a hierarchy selection value of 0x02 means that the accessory wants to navigate the video hierarchy.

Note: If the accessory sends a `ResetDBSelectionHierarchy` command while selecting the audio hierarchy, the database resets itself to the audio hierarchy and any video database selections are invalidated. Video selections already passed to the Playback Engine are unaffected.

Table 5-73 `ResetDBSelectionHierarchy` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x01 or 0x02	1	Hierarchy selection: 0x01 = audio, 0x02 = video.
Packet footer specified in "Command Packets" (page 109).		

Note: The Apple device will return an error if an accessory attempts to enable an unsupported hierarchy, such as a video hierarchy on an Apple device that does not support video.

5.1.55 Command 0x003C: GetDBiTunesInfo

Lingo: 0x04 — Origin: Accessory

Applies to Database Engine.

This command is sent by the accessory to get the specified Apple device iTunes database metadata information. In response, the Apple device sends a `RetDBiTunesInfo` command with the requested metadata.

Note: This DB metadata information is updated when the Apple device enters Extended Interface mode. This ensures that the information is updated following an iTunes sync event.

Table 5-74 `GetDBiTunesInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	iTunes database metadata type (see Table 5-75 (page 445))
Packet footer specified in "Command Packets" (page 109).		

Table 5-75 (page 445) shows the iTunes database metadata type codes.

Table 5-75 iTunes database metadata types

Type code	Description
0x00	Database UID unique to an Apple device and assigned by iTunes. This ID does not change when the Apple device is synced with iTunes nor when the Apple device's content is changed. It is not the same as the track UID returned by `GetDBTrackInfo` or `GetPBTrackInfo` and used by `GetUIDTrackInfo`.
0x01	Last sync date/time to iTunes on computer; updated when the Apple device is synced with iTunes even if no content has been added or deleted.
0x02	Total audio track count (including audio podcasts and audiobooks)
0x03	Total video track count (including movies, TV series, and video podcasts)
0x04	Total audiobook count
0x05	Total photo count
0x06-0xFF	Reserved

5.1.56 Command 0x003D: RetDBiTunesInfo

Lingo: 0x04 — Origin: Dev

Applies to Database Engine.

This command is returned by the Apple device in response to a `GetDBiTunesInfo` command received from the accessory. If the requested iTunes metadata information type is not within the valid range, an `iPodAck` command with a bad parameter status is returned.

Table 5-76 `RetDBiTunesInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	iTunes database metadata type (see Table 5-75 (page 445))
0x*NN*	*NN*	iTunes database metadata information (see Table 5-77 (page 445))
Packet footer specified in "Command Packets" (page 109).		

Table 5-77 (page 445) shows the iTunes database metadata information formats.

Table 5-77 iTunes database metadata information formats

Type code	Bytes	Description
0x00	8	iTunes database unique 64-bit identifier

Type code	Bytes	Description
0x01	7	Last sync date/time (see Table 5-78 (page 446))
0x02	4	Total audio track count
0x03	4	Total video track count
0x04	4	Total audiobook count
0x05	4	Total photo count
0x06-0xFF	Reserved	

Table 5-78 Date/time format

Byte	Description	Values
0	Seconds	00 - 59
1	Minute	00 - 59
2	Hour	00 = 12am, 23 = 11pm
3	Day	01 = 1st, 31 = 31st
4	Month	01 = Jan, 12 = Dec
5-6	Year	2007 = year 2007

5.1.57 Command 0x003E: GetUIDTrackInfo

Lingo: 0x04 — Origin: Accessory

Applies to Database Engine.

This command is sent by the accessory to get one or more types of track information using the track's Apple device-unique identifier. In response, the Apple device returns separate `RetUIDTrackInfo` commands for each type of track information requested. The track information mask is variable length; it is not necessary to transmit any trailing bytes in which no bits are set. If the track information mask contains any unrecognized track information type bits, a single `iPodAck` command is returned with a bad parameter status.

Note: The accessory's input buffer must be of sufficient size to accept all requested track information response packets without errors. The requested packets may be sent to the accessory in rapid succession, without any pauses or interruptions.

Table 5-79 `GetUIDTrackInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		

Value	Bytes	Parameter
0x*NN*	8	Unique track identifier; an accessory can obtain this value by sending a "Command 0x0040: GetDBTrackInfo" (page 451) or "Command 0x0042: GetPBTrackInfo" (page 452) command with bit 7 of byte 14 set to 1.
0x*NN*	*NN*	Track information type bitmask (see Table 5-80 (page 447)); do not set bit 7.
Packet footer specified in "Command Packets" (page 109).		

Table 5-80 (page 447) lists the track information type bitmask bits.

Table 5-80 Track information type bits

Bit	Description
0	Capabilities (media kind, skip when shuffle, has artwork, has bookmark, has lyrics, is audiobook, etc.)
1	Track name
2	Artist name
3	Album name
4	Genre name
5	Composer name
6	Total track time duration
7	Unique track identifier
8	Chapter count
9	Chapter times
10	Chapter names
11	Lyrics of the song currently playing in the Playback Engine
12	Description
13	Album track index
14	Disc set album index
15	Play count
16	Skip count
17	Podcast release date
18	Last played date/time
19	Year (release date)

Bit	Description
20	Star rating
21	Series name
22	Season number
23	Track volume adjust
24	Track EQ preset
25	Track data rate
26	Bookmark offset
27	Start/stop time offset
28...	Reserved

Note: Requesting the lyrics of a track not currently being played will result in a null string being returned. Lyrics for a track being played may take up to 5 seconds to return. Use `SetPlayStatusChangeNotification` to be notified when the lyrics for the currently playing track become available.

5.1.58 Command 0x003F: RetUIDTrackInfo

Lingo: 0x04 — Origin: Dev

Applies to Database Engine.

This command is sent by the Apple device in response to a `GetUIDTrackInfo` command received from the accessory. Multiple responses may be sent for each type of track information requested (e.g., chapter names).

Note: The accessory's input buffer must be of sufficient size to accept all requested track information response packets without errors. The requested packets may be sent to the accessory in rapid succession, without any pauses or interruptions.

Table 5-81 `RetUIDTrackInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	8	Unique track identifier
0x*NN*	*NN*	Track information type (bit position number)
0x*NN*	*NN*	Track information data (see Table 5-82 (page 449))
Packet footer specified in "Command Packets" (page 109).		

Table 5-82 (page 449) shows the track information data formats.

Table 5-82 Track information data formats

Type	Description	Bytes	Format
0	Capabilities	4	See Table 5-83 (page 450)
1	Track name	*NN*	Null-terminated UTF8 string
2	Artist name	*NN*	Null-terminated UTF8 string
3	Album name	*NN*	Null-terminated UTF8 string
4	Genre name	*NN*	Null-terminated UTF8 string
5	Composer name	*NN*	Null-terminated UTF8 string
6	Total track duration	4	Milliseconds
7	iTunes unique track ID	8	An 8-byte UID that uniquely identifies an Apple device track. This track UID is different from the iTunes database UID returned by `RetDBiTunesInfo`.
8	Chapter count	2	Chapter count (0 = no chapters)
9	Chapter times	2,4	2 bytes chapter index (0 = first chapter) followed by 4 bytes chapter offset in milliseconds from beginning of track. A separate index/offset pair is appended for each track chapter. If a track has no chapters, no data is returned.
10	Chapter names	2,*NN*	2 bytes chapter index (0 = first chapter) followed by the chapter name as null-terminated UTF8 string. A separate index/name pair is appended for each track chapter. If a track has no chapters, no data is returned.
11	Lyrics of the song currently playing in the Playback Engine	2,2,*NN*	2 bytes current track lyrics section index (0 = first section, 0x*NNNN* = last section index), followed by 2 bytes maximum track lyrics section index (0 = only 1 section, 0x*NNNN* = maximum section index), followed by some or all of the track lyrics string. The track lyrics as a whole consists of a single null-terminated UTF8 string. If the lyrics string is too long to be carried in a single packet, then the string is broken into sections and carried in separate packets, with different values for each section index. The last lyrics packet section contains the null terminator character for the full string. Lyrics sections before the last section do not include null terminators and may not be valid UTF8 strings. Accessories must use the packet payload length to determine the length of the track lyrics string and assemble it by concatenating its substrings in order. Requesting the lyrics of a track not currently being played will result in a null string being returned.
12	Description	*NN*	Null-terminated UTF8 string

Type	Description	Bytes	Format
13	Album track index	2	index number
14	Disc set album index	2	index number
15	Play count	4	Track play count (0 = track not played)
16	Skip count	4	Track skip count (0 = track not skipped)
17	Podcast release date	7	Date/time (see Table 5-78 (page 446))
18	Last played date/time	7	Date/time (see Table 5-78 (page 446)); all zeroes if the track has never been played.
19	Year (release date)	2	Year in which track was released
20	Star rating	1	Star rating of track; 00 = No stars, 20 = 1 star, 40 = 2 stars, 60 = 3 stars, 80 = 4 stars, 100 = 5 stars.
21	Series name	*NN*	Null-terminated UTF8 string
22	Season number	2	Season number (1 = First season)
23	Track volume adjust	1	Track volume attenuation/amplification adjustment (0x9C = –100%, 0x00 = no adjust, 0x64 = +100%)
24	Track EQ preset	2	Track equalizer preset index
25	Data rate	4	Track bit rate (kilobits per second)
26	Bookmark offset	4	Bookmark offset from start of track in milliseconds
27	Start/stop time offset	4,4	4 bytes start time followed by 4 bytes stop time in milliseconds
28...	Reserved		

Table 5-83 Capabilities bits

Bit	Description
00	1 = Is audiobook
01	1 = Has chapters
02	1 = Has artwork
03	1 = Has lyrics
04	1 = Is podcast episode
05	1 = Has release date
06	1 = Has description

Bit	Description
07	1 = Is video
08	1 = Is queued as video (in the Playback Engine only, not in the database)
12:09	Reserved
13	1 = Is capable of generating a Genius playlist
14	1 = Is an iTunesU episode
31:15	Reserved

5.1.59 Command 0x0040: GetDBTrackInfo

Lingo: 0x04 — Origin: Accessory

Applies to Database Engine.

This command is sent by the accessory to get the specified Apple device database track information types for the specified track index range. In response, the Apple device returns separate `RetDBTrackInfo` packets for each type of track information requested. The starting track index is based on the current database track selection(s). A track count of 0xFFFFFFFF (–1) returns the track information for all Apple device tracks from the starting DB track index. The track information mask is variable length; it is not necessary to transmit any trailing bytes in which no track information bits are set. If the track index, count, or information mask contains any invalid data, a single `iPodAck` command is returned with a bad parameter status.

Note: The accessory's input buffer must be of sufficient size to accept all requested track information response packets without errors. The requested packets may be sent to the accessory in rapid succession, without any pauses or interruptions.

Table 5-84 `GetDBTrackInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Track database start index
0x*NN*	4	Track count (from track start index)
0x*NN*	*NN*	Track information type bitmask (see Table 5-80 (page 447))
Packet footer specified in "Command Packets" (page 109).		

Note: Requesting the lyrics of a track not currently being played will result in a null string being returned. Lyrics for a track being played may take up to 5 seconds to return. Use `SetPlayStatusChangeNotification` to be notified when the lyrics for the currently playing track become available.

5.1.60 Command 0x0041: RetDBTrackInfo

Lingo: 0x04 — Origin: Dev

Applies to Database Engine.

This command is sent by the Apple device in response to a `GetDBTrackInfo` command received from the accessory. Multiple responses may be sent for each type of track information requested (e.g., chapter names).

Note: The accessory's input buffer must be of sufficient size to accept all requested track information response packets without errors. The requested packets may be sent to the accessory in rapid succession, without any pauses or interruptions.

Table 5-85 `RetDBTrackInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Track database index
0x*NN*	1	Track information type (bit position number)
0x*NN*	*NN*	Track information data (see Table 5-82 (page 449))
Packet footer specified in "Command Packets" (page 109).		

5.1.61 Command 0x0042: GetPBTrackInfo

Lingo: 0x04 — Origin: Accessory

Applies to Playback Engine.

This command is sent by the accessory to get the specified Apple device playing track information types for the specified track index range. In response, the Apple device returns separate `RetPBTrackInfo` packets for each type of track information requested. A track count of 0xFFFFFFFF (–1) returns the track information for all Apple device tracks from the starting PB track index. The track information mask is variable length; it is not necessary to transmit any trailing bytes in which no track information bits are set. If the track index, count, or information mask contains any invalid data, a single `iPodAck` command is returned with a bad parameter status.

Note: The accessory's input buffer must be of sufficient size to accept all requested track information response packets without errors. The requested packets may be sent to the accessory in rapid succession, without any pauses or interruptions.

Table 5-86 `GetPBTrackInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Track playing start index
0x*NN*	4	Track count (from track start index)
0x*NN*	*NN*	Track information type bitmask (see Table 5-80 (page 447))
Packet footer specified in "Command Packets" (page 109).		

Note: Requesting the lyrics of a track not currently being played will result in a null string being returned. Lyrics for a track being played may take up to 5 seconds to return. Use `SetPlayStatusChangeNotification` to be notified when the lyrics for the currently playing track become available.

5.1.62 Command 0x0043: RetPBTrackInfo

Lingo: 0x04 — Origin: Dev

Applies to Playback Engine.

This command is sent by the Apple device in response to a `GetPBTrackInfo` command received from the accessory. Multiple responses may be sent for each type of track information requested (e.g., chapter names).

Note: The accessory's input buffer must be of sufficient size to accept all requested track information response packets without errors. The requested packets may be sent to the accessory in rapid succession, without any pauses or interruptions.

Table 5-87 `RetPBTrackInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Track playback index
0x*NN*	1	Track information type (bit position number)
0x*NN*	*NN*	Track information data (see Table 5-82 (page 449))
Packet footer specified in "Command Packets" (page 109).		

5.1.63 Command 0x0044: CreateGeniusPlaylist

Lingo: 0x04 — Origin: Accessory

Applies To: Database Engine and Playback Engine

The accessory sends this command to an Apple device to ask it to create a Genius playlist. The Apple device returns an Extended Interface `iPodAck` command, passing one of the responses listed in Table 4-98 (page 282). If the playlist is successfully created, it starts playing.

Note: This command may take up to 30 seconds to finish.

Table 5-88 `CreateGeniusPlaylist` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Index type; see Table 5-89 (page 454).
0x*NN*	4	Track index
Packet footer specified in "Command Packets" (page 109).		

Table 5-89 Index types

Value	Description
0x00	Database Engine
0x01	Playback Engine
0x02-0xFF	Reserved

5.1.64 Command 0x0045: RefreshGeniusPlaylist

Lingo: 0x04 — Origin: Accessory

Applies To: Database Engine

The accessory sends this command to an Apple device to ask it to refresh a Genius playlist previously created by `CreateGeniusPlaylist`. The Apple device returns an Extended Interface `iPodAck` command, passing one of the responses listed in Table 5-91 (page 455).

Notes: A Genius playlist should be refreshed only after it has been selected during database browsing. This command may take up to 30 seconds to finish.

Table 5-90 `RefreshGeniusPlaylist` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Playlist index
Packet footer specified in "Command Packets" (page 109).		

Table 5-91 iPodAck responses to RefreshGeniusPlaylist

iPodAck response	Meaning
0x00	Genius playlist being refreshed. When the refresh is complete, the Apple device sends an `iPodNotification` command for Command Complete; see Table 3-124 (page 189).
0x02	Genius playlist could not be refreshed.
0x12	Playlist is not a Genius playlist.

5.1.65 Command 0x0047: IsGeniusAvailableForTrack

Lingo: 0x04 — Origin: Accessory

Applies To: Database Engine and Playback Engine

The accessory sends this command to an Apple device to determine if Genius information is available for a given track. The Apple device returns an Extended Interface `iPodAck` command, passing one of the responses listed in Table 4-100 (page 283).

Table 5-92 `IsGeniusAvailableForTrack` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Index type; see Table 5-89 (page 454).
0x*NN*	4	Track index
Packet footer specified in "Command Packets" (page 109).		

5.1.66 Command 0x0048: GetPlaylistInfo

Lingo: 0x04 — Origin: Accessory

Applies To: Database Engine

The accessory sends this command to an Apple device to get information about a given playlist. Currently the only information type is playlist capabilities (0x00). The Apple device returns a `RetPlaylistInfo` command, passing zero or more of the bits listed in Table 5-94 (page 456).

Table 5-93 `GetPlaylistInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Info type; see Table 5-94 (page 456).
0x*NN*	4	DB playlist index
Packet footer specified in "Command Packets" (page 109).		

Table 5-94 GetPlaylistInfo info types

Info type value	Description	Data length
0x00	Playlist information: Bit 0: Is Genius playlist Bits 1-31: Reserved	4 bytes
0x01-0xFF	Reserved	

5.1.67 Command 0x0049: RetPlaylistInfo

Lingo: 0x04 — Origin: Apple device

Applies To: Database Engine

The Apple device sends this command to an accessory in response to a `GetPlaylistInfo` command.

Table 5-95 `RetPlaylistInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Info type; see Table 5-94 (page 456).
0x*NN*	1	Info data; see Table 5-94 (page 456).

Value	Bytes	Parameter
Packet footer specified in "Command Packets" (page 109).		

5.1.68 Command 0x004A: PrepareUIDList

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

The accessory sends this command to the Apple device to prepare a list of tracks for playback. The payload data consists of a list of track UIDs, each of which can be retrieved using the `GetDBTrackInfo` command. Because the list of UIDs can be larger than the maximum payload size that the Apple device supports for the transport, this command is a multisection command. The accessory must send a `RequestTransportMaxPayloadSize` command to determine the maximum payload size and then follow the procedure described in "Multisection Data Transfers" (page 116). All sections of the command must have the same transaction ID.

If the accessory sends a new list of trackUIDs before finishing the transfer of the old list, the transfer of the old list will be stopped and further sections for the previous command will be acknowledged with a Bad Parameter error. Sending another `PrepareUIDList` command with an empty payload clears the existing prepared UID list.

The prepared UID list is maintained until the next `PrepareUIDList` command is received or Extended Interface mode is exited. The order of track UIDs in the list is preserved through multiple playbacks unless shuffle is enabled.

Table 5-96 `PrepareUIDList` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	2	`sectCur` ; see "Multisection Data Transfers" (page 116).
0x*NN*	2	`sectMax`
0x*NN*	*NN*	Sequence of 64-bit `trackUID` values; see "Command 0x0040: GetDBTrackInfo" (page 451).
Packet footer specified in "Command Packets" (page 109).		

5.1.69 Command 0x004B: PlayPreparedUIDList

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

The Accessory sends this command to start playback of the list of tracks prepared by a previous `PrepareUIDList` command. Its parameters specify the `trackUID` of the track from which to start playback and when to start playback of the new list. This command loads only the track UID list established by `PrepareUIDList` into the playback queue. If there is no such list, because `PrepareUIDList` was never sent or it failed, this command is acknowledged with a Command Failed error.

The `trackUID` parameter specifies which track in the prepared list to play first. If the specified UID is not valid or not in the prepared track UID list, then the first track in the list is played first.

If Shuffle Mode is enabled, this command causes the track specified by the `trackUID` parameter to be put at the beginning of the playback list, and the rest of the tracks are then shuffled behind that track.

Table 5-97 `PlayPreparedUIDList` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x00	1	Reserved
0x*NN*	8	`trackUID`: UID of the track in the prepared list to play first.
Packet footer specified in "Command Packets" (page 109).		

5.1.70 Command 0x004C: GetArtworkTimes

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

The accessory sends this command to the Apple device to request the list of artwork time locations for a track. The track can be specified by its UID, or its index in the playback engine queue list, or its index in the database selection list, depending on the value passed in the `trackIdentifierType` parameter.

A 2-byte `formatID` indicates which type of artwork is desired. See "Transferring Album Art" (page 64).

The 2-byte `artworkIndex` specifies at which index to begin searching for artwork. A value of 0 indicates that the Apple device should start with the first available artwork.

The 2-byte `artworkCount` specifies the maximum number of times (artwork locations) to be returned. A value of –1 (0xFFFF) indicates that there is no preferred limit. Note that podcasts may have a large number of associated images.

Table 5-98 `GetArtworkTimes` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`trackIdentifierType`: 0x00 = UID, 0x01 = playback list index, 0x02 = database index.
0x*NN*	*NN*	`trackIdentifier`: 8 bytes UID or 4 bytes index.

Value	Bytes	Parameter
0x*NN*	2	`formatID`
0x*NN*	2	`artworkIndex`
0x*NN*	2	`artworkCount`
Packet footer specified in "Command Packets" (page 109).		

5.1.71 Command 0x004D: RetArtworkTimes

Lingo: 0x04 — Origin: Apple device

Applies To: Playback Engine

The Apple device sends this command to the accessory to return the list of artwork times for a given track. The Apple device returns zero or more 4-byte times, one for each piece of artwork associated with the track and format specified by `GetArtworkTimes`. See "Transferring Album Art" (page 64).

The number of records returned will be no greater than the number specified in the `GetArtworkTimes` command. It may, however, be less than requested. This can happen if there are fewer pieces of artwork available than were requested, or if the Apple device is unable to place the full number in a single packet.

Table 5-99 `RetArtworkTimes` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`trackIdentifierType`: 0x00 = UID, 0x01 = playback list index, 0x02 = database index.
0x*NN*	*NN*	`trackIdentifier`: 8 bytes UID or 4 bytes index.
0x*NN*	*NN*	Sequence of 4-byte `timeOffset` records, each specifying a time in ms from the start of the track.
Packet footer specified in "Command Packets" (page 109).		

5.1.72 Command 0x004E: GetArtworkData

Lingo: 0x04 — Origin: Accessory

Applies To: Playback Engine

The accessory sends this command to the Apple device to request data for a given track, `formatID`, and `artworkIndex`. See "Transferring Album Art" (page 64). The track can be specified by its UID, or its index in the playback engine queue list, or its index in the database selection list, depending on the value passed in the `trackIdentifierType` parameter. The time offset from track start is the value returned by `RetArtworkTimes`.

Table 5-100 `GetArtworkData` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	`trackIdentifierType`: 0x00 = UID, 0x01 = playback list index, 0x02 = database index.
0x*NN*	*NN*	`trackIdentifier`: 8 bytes UID or 4 bytes index.
0x*NN*	2	`formatID`
0x*NN*	4	time offset from track start, in ms
Packet footer specified in "Command Packets" (page 109).		

5.1.73 Command 0x004F: RetArtworkData

Lingo: 0x04 — Origin: Apple device

Applies To: Playback Engine

The Apple device sends the requested artwork to the accessory. Multiple `RetArtworkData` commands may be necessary to transfer all the data because it will be too much to fit into a single packet. See "Transferring Album Art" (page 64).

This command uses nearly the same format as the `SetDisplayImage` command (command 0x0032). The only difference is the addition of 2 coordinates; they define an inset rectangle that describes any padding that may have been added to the image. The coordinates consist of two x,y pairs. Each x or y value is 2 bytes, so the total size of the coordinate set is 8 bytes.

Table 5-101 `RetArtworkData` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	2	`sectCur` ; see "Multisection Data Transfers" (page 116).
0x*NN*	2	`sectMax`
0x*NN*	*NN*	Artwork data payload: sequence of artwork data sections (see below).
Packet footer specified in "Command Packets" (page 109).		

The artwork data payload returned by `RetArtworkData` starts with a 1-byte `trackIdentifierType` value: 0x00 = UID, 0x01 = playback list index, 0x02 = database index. If the payload is sent in multiple packets, this value will start the payload region of only the first packet.

The payload formats for different `trackIdentifierType` values are shown in Table 5-103 (page 461) and Table 5-103 (page 461).

Table 5-102 `RetArtworkData` payload for `trackIdentifierType` = 0x00

Byte number	Value	Description
0	0x00	`trackIdentifierType`
1-8	0x*NN*	`trackUID`: UID of the track for this artwork.
9-*NN*	0x*NN*	`imageDescriptionAndData`; see Table 5-104 (page 461).

Table 5-103 `RetArtworkData` payload for `trackIdentifierType` = 0x01 or 0x02

Byte number	Value	Description
0	0x01 or 0x02	`trackIdentifierType`
1-4	0x*NN*	`trackIndex`: Index of the track for this artwork.
5-*NN*	0x*NN*	`imageDescriptionAndData`; see Table 5-104 (page 461).

Table 5-104 `imageDescriptionAndData` format

Byte	Value	Description
0	0x*NN*	`pixelFormatCode`
1	0x*NN*	Image width in pixels (15:8)
2	0x*NN*	Image width in pixels (7:0)
3	0x*NN*	Inset rectangle, top-left point, x value (15:8)
4	0x*NN*	Inset rectangle, top-left point, x value (7:0)
5	0x*NN*	Inset rectangle, top-left point, y value (15:8)
6	0x*NN*	Inset rectangle, top-left point, y value (7:0)
7	0x*NN*	Inset rectangle, bottom-right point, x value (15:8)
8	0x*NN*	Inset rectangle, bottom-right point, x value (7:0)
9	0x*NN*	Inset rectangle, bottom-right point, y value (15:8)
10	0x*NN*	Inset rectangle, bottom-right point, y value (7:0)
11	0x*NN*	Row size in bytes (31:24)
12	0x*NN*	Row size in bytes (23:16)
13	0x*NN*	Row size in bytes (15:8)
14	0x*NN*	Row size in bytes (7:0)

Byte	Value	Description
15-*NN*	0x*NN*	Image pixel data (variable length)

5.2 Known Issues

- Apple devices may take up to 20 seconds to detect that the remote accessory has been detached from the 30-pin connector. This depends on the firmware version and whether or not power is being supplied to the Apple device.
- External accessories such as simple remote controls may still be active, even when the Apple device has entered Extended Interface mode. Users should be discouraged from attaching any accessories to the Apple device while it is in Extended Interface mode.
- iPod mini version 1.4 and 4G iPod version 3.1 do not return audiobook chapter names. They return an empty string instead.

5.3 Protocol History

The following is a history of the changes made to Extended Interface mode.

Table 5-105 History of Extended Interface mode

Version	Changes
Version 1.14	Added status change control extensions to `SetPlayStatusChangeNotification` and `PlayStatusChangeNotification`.
	Added new support for Next Chapter and Previous Chapter in the `PlayControl` command.
	Added new commands 0x0044-0x0045 and 0x0047 through 0x0049.
Version 1.13	Added support for nested playlists.
	Added Play and Pause control codes to command 0x0029.
	Added new commands 0x003C through 0x004F.
Version 1.12	Improved video browsing support for `SelectDBRecord(−1)`.
Version 1.11	This version adds `ResetDBSelectionHierarchy` command to enable video browsing.
	New feature: video browsing.
	Chapter information can be retrieved for all tracks in the Now Playing list, not just for the currently playing track.

Version	Changes
Version 1.10	This version adds commands 0x000E through 0x0011 and 0x002A through 0x002B.
	New features: Code restructured to improve performance.
	Added indexed playing track genre and composer info.
	Added playing track artwork support.
	Bug fixes: Return track description and lyrics info if present.
	Notify track changes even when playback is paused.
Version 1.09	This version adds support for color images, podcast chapters, and fixes a few bugs:
	`SetDisplayImage` supports color images on color Apple devices.
	A bug that caused `SelectDBRecord` to fail if there were no podcasts on the Apple device is fixed.
	A bug that caused `SelectDBRecord` and `SelectSortDBRecord` to reverse track order after a podcast was selected is fixed.
	`PlayStatusChange` notifications support podcast chapters.
	`SetCurrentPlayingChapter` now accepts a chapter index for podcasts.
	Added new commands `GetColorDisplayImageLimits` and `ReturnColorDisplay-ImageLimits` to obtain information about color display images.
Version 1.08	This version adds support for the Podcast category, chapter names, and indexed playing track information. It also includes a bug fix for `SetDisplayImage`.
Version 1.07	This version adds the restore on exit feature to the set shuffle, set repeat, and set audiobook speed setting commands. It also fixes a few bugs in the database engine management commands.
	`ResetDBSelection` clears the database sort order.
	`SelectDBRecord` with an invalid index returns a command error.
	Audiobook speed restore on exit with optional flag.
	Shuffle setting restore on exit with optional flag.
	Repeat setting restore on exit with optional flag.
Version 1.06	This version adds several new lingo 0x04 commands to allow remote accessories to control audiobook playback. Please refer to lingo 0x04 commands 0x0002 through 0x000B for details. The following Extended Interface commands have been modified to support audiobook playback:
	`SelectDBRecord`

Version	Changes
	`GetNumberCategorizedDBRecords`
	`ReturnNumberCategorizedDBRecords`
	`RetrieveCategorizedDatabaseRecords`
	`ReturnCategorizedDatabaseRecord`
	`PlayStatusChangeNotification`
	`PlayControl`
	`SelectSortDBRecord`. The sort order field is ignored for the case of an audiobook, as audiobooks are automatically sorted by name.
Version 1.05	This version adds several new lingo 0x00 commands and minor fixes to version 1.04. The changes include:
	Fixed a problem where iPod Accessory Protocol commands on the 30-pin connector did not wake up unpowered, sleeping Apple devices.
	Fixed the `SelectDBRecord` routine to play tracks from a stopped play state.
	Fixed `RetrieveCategorizedDBRecords` to retrieve all records from the start index when passed a count of -1.
	Fixed long record names resulting in large packets causing the Apple device to reboot.
	Added missing begin and end FF or REW notification messages.
Version 1.04	This version adds some minor fixes to Version 1.03. The changes include:
	Fixed a problem where accessories were intermittently not detected when the 30-pin connector was plugged in to the Apple device.
	Optimized the Apple device `ReturnPlayStatus` response to the `GetPlayStatus` message from the accessory. Command response is much faster.
Version 1.03	This version is the functional equivalent of protocol version 1.02 (1.03 is equal to 1.02). It is only reported on the iPod mini in software version 1.1. All of the changes reported for version 1.02 are applicable for 1.03 and no more.
Version 1.02	This version was released in the third generation (3G) iPod with firmware version 2.2. The changes include:
	Fixed a problem where the user's On-The-Go playlist was erased when Extended Interface mode was initialized.
	Fixed a problem where a playing Apple device was stopped and playing state lost when the Extended Interface mode was initialized.
	Fixed a problem where the strings returned from the Apple device were truncated if there were multibyte characters present.

Version	Changes
	Fixed a problem where an Apple device playing a track in Extended Interface mode would continue playing the same track, unpaused, when it was disconnected and reverted to the Apple device UI mode.
	Fixed a problem where an Apple device in Extended Interface mode failed to go into Sleep mode when power was removed from it.
	Fixed a problem where an Apple device with a set alarm would honor the alarm while in Extended Interface mode or soon after disconnection.
	Fixed a problem where the pause indicator was shown when playback was stopped during Extended Interface mode.
Version 1.01	This version was released in the iPod mini version 1.0 system software with bug fixes and protocol improvements during February 2004. The changes include:
	Added the commands: `SelectSortDBRecord`, `SetCurrentPlayingTrack`, `GetNumPlayingTracks`, `GetMonoDisplayImageLimits` and `SetDisplayImage`.
	Changed the default sort order of returned or listed database items to match that of the Apple device UI.
	Fixed a transmit problem where large packets being sent by the Apple device would pause, mid-packet, until the beginning of another packet was received by the Apple device.
	Fixed a problem where the Apple device play/pause button remained active when Extended Interface mode was in control of the Apple device.
Version 1.0	This is the base protocol released in October 2003 in system software version 2.1 of the 3G iPod.

5. Extended Interface Mode

A. iTunes Tagging Accessory Design

Apple's iTunes Tagging feature is described in "iTunes Tagging" (page 83). This appendix specifies how broadcast reception accessories must be designed to support this feature.

A.1 Accessory Requirements

Broadcast reception accessories that support the iTunes tagging feature described in this specification must perform the following actions:

- Authenticate themselves to the attached Apple device. This requires that the accessory contain an authentication coprocessor supplied by Apple. For technical details, see Apple's *iPod Authentication Coprocessor 2.0B Specification*.
- Conform to the user interface requirements described in "Accessory User Interface" (page 474).
- Provide static storage capacity for at least 50 tags, and provide enough RAM to cache two tags in cases of tag ambiguity (see "Resolving Tag Ambiguity" (page 468)). Accessories should store tags in a data structure and only generate XML when writing files to the Apple device. Tag data may be stored internally in any format convenient to the accessory.
- Generate tag data from the broadcast's metadata. Accessories supporting iTunes Tagging must populate every XML field for which metadata is received.
- Generate files from the tag data and write these files to the attached Apple device. The files must be XML-formatted **plists,** as specified in "Tag Data Writing Process" (page 469). The plist fields in these files are specified in "Data Transfer to the Apple Device" (page 471), and the iAP commands used to write a file to the Apple device are defined in "Lingo 0x0C: Storage Lingo" (page 357).

A.2 Capturing and Storing Tag Data

The process of capturing and storing tag data from a broadcast occurs in two steps:

1. When the user presses the tag button while listening to the radio accessory, the accessory stores tag data internally.
2. When the user attaches an Apple device to the radio accessory the stored tag data is written to the Apple device's memory.

Note: Tag data must be written to the Apple device whenever the Apple device is attached, regardless of what mode the accessory may be in. The user must never have to tell the accessory to write tag data nor have to change the accessory's mode to permit tag data to be written.

A.2.1 Resolving Tag Ambiguity

A broadcast tag is deemed ambiguous if the tag button is pressed within 10 seconds before or after a change in the program metadata. In an HD broadcast, a program metadata change is indicated by a Program Service Data (PSD) change; in an FM broadcast, it is indicated by the Event A/B flag. This ambiguity period exists because program metadata changes do not occur exactly on song boundaries, but within 10 seconds before or after that boundary. If the button is pressed less than 10 seconds before the program metadata changes and the new program metadata data is not yet valid, the receiver should generate a nonambiguous tag for the previous data. If a nonambiguous tag is followed by an ambiguous tag of the next song, the result should be two nonambiguous tags. The accessory's UI should not tell the user that a tag is ambiguous.

Note: Tag ambiguity never happens when the user is tagging satellite radio broadcasts.

The accessory must flag ambiguous tags using the `ambiguousTag` plist field. The iTunes application will present a user interface dialog to resolve the ambiguity once the tags have been imported. Accessories should not try to resolve tag ambiguity through their own user interface. Accessories must identify each twin of the ambiguous pair and save this information as a part of the tag data in RAM. Accessories must also identify which twin was active at the time of the tag button press by setting the `ButtonPressed` plist field.

To support tag ambiguity, accessories must store both the current and previous sets of tag data in RAM and update both as program metadata changes occur. A timestamp or timer can be used to mark when either the program metadata changes or the tag button is pressed. The previous and current tags should be stored (when no Apple device is connected) or written to the Apple device with the `ambiguousTag` plist field set to 1 if these two events occur within 10 seconds of each other.

If the tag button is pressed within 10 seconds **before** a program change:

- The accessory timestamps the button press, or starts a 10 second timer when the tag button is pressed, and sets the `ButtonPressed` byte in the current tag data.
- If a program change occurs within the 10 seconds, the tag data in the current slot is moved to the previous slot and the broadcast tag data fills the current slot. The accessory sets the `ambiguousTag` fields for both tags to 1 and writes both tags to the Apple device in a single file.

If the tag button is pressed within 10 seconds **after** a program change:

- The accessory timestamps the program change event, or starts a 10 second timer when the program change occurs, and updates the tag data in the current and previous slots.
- If the tag button is pressed within 10 seconds of the program change, the accessory sets the `ButtonPressed` byte in the current tag data, sets the `ambiguousTag` fields for both tags to 1, and writes both tags to the Apple device in a single file.

Note: Accessories must resolve tag ambiguity (if it exists) before saving tags or writing them to the Apple device. This adds a 10 second delay to the file writing process.

A.2.2 Handling Unknown Metadata Types

Accessories must support features announced after their product releases by supporting the `UnknownData` plist field. This field is used to pass to iTunes unrecognized data types in the broadcast metadata; iTunes parses the data in the `UnknownData` field and handles it appropriately. In HD broadcasts, unknown metadata may occur in the PSD's Unique File Identifier Data (UFID) fields; in FM broadcasts, it may be passed by the iTunes ODA. On encountering unrecognized ID types in a broadcast, iTunes Tagging accessories must encode the entire unknown data field (the complete UFID in the case of HD and the whole iTunes ODA in the case of FM) in `base64` format (64 bytes) and write it to the Apple device as `UnknownData`.

The current HD radio UFID data ID types are listed in Table A-1 (page 469). The current FM tag element types are listed in Table A-9 (page 482).

Table A-1 HD UFID data ID types

ID type	Description
0x3031 or "01"	`iTunesSongID`
0x3032–0x3033 or "02"–"03"	Reserved
0x3034 or "04"	`iTunesAffiliateID`
0x3035 or "05"	`iTunesStorefrontID`
0x3036 or "06"	The contents of the `stationURL` field is a `PodcastFeedURL`, not a station URL.

Note: The data in the HD radio UFID Identifier field is encoded in conformance with specification ISO-8859-1 and is represented as strings. Refer to iBiquity Document 2174 for more information about parsing the HD UFID data field.

A.2.3 Tag Data Writing Process

Each tagged track file written to an Apple device consists of an extensible XML dictionary of key-value pairs, formatted as a Mac OS X Core Foundation property list (**plist**). This format is widely used in Mac OS X and can be easily generated and parsed on other platforms. The format is documented at developer.apple.com/documentation/CoreFoundation/Conceptual/CFPropertyLists/index.html. The plist document type definition (DTD) is available at www.apple.com/DTDs/PropertyList-1.0.dtd. Listings of typical of plist files can be found in "Sample Tag Files" (page 492).

Note: Mac OS X plist files are UTF-8 encoded; the PSD data sent by HD radio broadcasters is ISO-8859-1 encoded. The radio accessory must convert the IS0-8859-1 data to UTF-8 before writing the plist file to the Apple device. If the plist file contains ISO-8859-1 characters in the range 0x80 to 0xFF, iTunes cannot parse the file properly and the tag will be lost. The XML markup delimiters &, <, and > must be written as `&`, `<`, and `>`—otherwise iTunes cannot open the file.

After authenticating itself to the attached Apple device and using `GetiPodOptionsForLingo` to determine that the Apple device supports the iTunes Tagging option, an accessory typically uses the command sequence shown in Table A-2 (page 470) to write a plist file to the Apple device. For details of these commands, see "Lingo 0x0C: Storage Lingo" (page 357). For a sample sequence of actual commands, see Table A-6 (page 476).

Table A-2 Typical plist writing command sequence

Radio accessory	Attached Apple device
`GetiPodCaps`	
	`RetiPodCaps`
`GetiPodFreeSpace`	
	`RetiPodFreeSpace`
`OpeniPodFeatureFile`	
	`RetiPodFileHandle`
`WriteiPodFileData` (as many times as needed)	
	`iPodAck` for each `WriteiPodFileData` command
`CloseiPodFile`	
	`iPodAck` of `CloseiPodFileData`

The accessory starts by sending a `GetiPodCaps` command to retrieve the Apple device's storage lingo capabilities. The Apple device responds with the `RetiPodCaps` command, which contains the following data:

- `totalSpace`: the amount of free storage on the Apple device.
- `maxFileSize`: the largest possible size of a file on the Apple device.
- `maxWriteSize`: the largest amount of data that can be written to the Apple device in a single `WriteiPodFileData` command.
- `majorVersion` and `minorVersion`: the version number of the Storage lingo protocol (currently 1.1).

Accessories are required to honor the Apple device's `maxWriteSize` limitation; they must never send a `WriteiPodFileData` command with a data payload larger than the Apple device's `maxWriteSize` value.

The accessory should also check the Apple device's free space before creating a file, to ensure that there is enough free space to write the file. A tag takes approximately 1 KB of data space, so the accessory should verify that the Apple device has 1 KB of free space for each tag is intends to write. See "Note" (page 359). The accessory should also warn the user through its user interface when the Apple device does not have enough free space; see "Accessory User Interface" (page 474) for details.

To create a file on the Apple device, the accessory sends an `OpeniPodFeatureFile` command. This command returns a file handle that is used to write data and to close the file. The accessory must send a `WriteiPodFileData` command to write data to the open file. The size of the `WriteiPodFileData` payload is determined by the Apple device's `maxWriteSize`; it may take several `WriteiPodFileData` commands to write an entire file. The Apple device responds to each `WriteiPodFileData` command with an `iPodAck` command containing the status of the last write. Accessories must verify that each individual write succeeded before sending the next `WriteiPodFileData` command.

Only one tagging file may be opened at a time. Each time a tagging file is opened, a new file is created in `/iPod_Control/Device/Accessories/Tags/` on the Apple device (this location may change in future releases). You can look at the files in this directory to confirm that the data you wrote using `WriteiPodFileData` was transferred correctly.

The accessory should close the file using the `CloseiPodFile` command as soon as all the plist data has been written to the Apple device. The accessory should verify that the Apple device closed the file properly by checking the status of the `iPodAck` command sent in response to the `CloseiPodFile` command. Accessories should never delete tag data stored locally until the `CloseiPodFile` acknowledgment status has been verified.

A.2.4 Data Transfer to the Apple Device

Table A-3 (page 471) shows all the plist fields an accessory may write to an Apple device. Subsets of these fields for specific broadcasting types are listed in "FM Radio Tagging" (page 480), "HD Radio Tagging" (page 490), "Satellite Radio Tagging" (page 492), and "Other Broadcast Tagging" (page 492).

The plist writing process must follow these rules:

- All fields of integer type must be expressed in decimal numbers. Hexadecimal and other non-decimal numbers may not be used.
- Strings in the `Name`, `Artist`, `Album`, `Genre`, and `StationURL` fields must not be truncated; if 64 bytes are received, all 64 must be written. The `StationURL` field may contain 128 bytes.

Table A-3 Plist fields written to the Apple device

Name	Type	Description	Source
`MajorVersion`	integer	The major revision level of the file format. This number increments by 1 if there are changes or additions that break compatibility with all prior versions. The current major revision level is 1.	Accessory

Name	Type	Description	Source
`MinorVersion`	integer	The minor revision level of the file format. This number increments by 1 if there are changes or additions that preserve compatibility with prior versions that have the same major revision level. The current minor revision level is 1.	Accessory
`ManufacturerID`	positive integer (decimal format)	A product-specific ID number (see note, below). This number is different from the number assigned to Nike + iPod Cardio accessories, described in Table B-5 (page 509).	Accessory
`ManufacturerName`	string	The manufacturer-assigned user-visible name for the manufacturer.	Accessory
`DeviceName`	string	The manufacturer-assigned user-visible name for the accessory.	Accessory
`MarkedTracks`	array	An array of one or more dictionaries, specifying the data for each tagged track.	Accessory
`AmbiguousTag`	integer	1 to mark an ambiguous track; 0 otherwise.	Accessory
`ButtonPressed`	integer	1 to designate which twin in an ambiguous pair was active when the tag button was pressed; 0 otherwise.	Accessory
`Name`	string	The name of the track.	Broadcast Metadata
`Artist`	string	The name of the artist.	Broadcast Metadata
`Album`	string	The name of the album.	Broadcast Metadata
`Genre`	string	The genre of the track.	Broadcast Metadata
`iTunesSongID`	integer	The iTunes song identifier.	Broadcast Metadata
`iTunesStorefrontID`	integer	The iTunes storefront identifier.	Broadcast Metadata
`StationFrequency`	string	The station's frequency, expressed only by digits and an optional decimal point.	Accessory
`StationCallLetters`	string	The station identifier.	Broadcast Metadata

Name	Type	Description	Source
TimeStamp	date	The date and time at which the tagged track was broadcast, in ASCII timestamp format (e.g., 2008-04-16T00:35:42Z). The time zone must be UTC; iTunes converts it into local time.	Broadcast Metadata
PodcastFeedURL	integer	A value of 1 if the StationURL points to a podcast, 0 otherwise.	Broadcast Metadata
StationURL	string	The URL of the station, or of a podcast if PodcastFeedURL is 1.	Broadcast Metadata
ProgramNumber	integer	The station's multicast channel in the range 1 to 8, where 1 indicates the main program.	Broadcast Metadata
iTunesAffiliateID	string	The iTunes affiliate ID, as broadcast (not applicable to FM broadcasts—leave blank).	Broadcast Metadata
iTunesStationID	string	The iTunes station ID, as broadcast.	RDS only
UnknownData	data	A field used to pass unknown UFID data to iTunes, defined only for HD and FM broadcasts. See "Handling Unknown Metadata Types" (page 469).	Broadcast Metadata

Note: The ManufacturerID number is part of a unique product ID assigned by Apple. The first part of the Apple-assigned ID identifies the licensee's account and the second part is the specific product number to be entered in this field. For example, if Apple assigns a product ID of 678999-12345, the ManufacturerID field must be set to 12345. To obtain a product ID number, the licensee must submit a product plan to Apple's MFi portal. Apple will respond by assigning a unique number to every separate product (i.e., every SKU).

The radio accessory normally writes a tag to the Apple device each time the user presses the tag button. However, the tag should not be written if tag ambiguity exists and has not been resolved (see "Resolving Tag Ambiguity" (page 468)). The opening, writing, and closing of the file should happen in one continuous action to insure that no tag information is lost. Accessories should not keep files open any longer than absolutely necessary. Do not append tags to open files; write a new file to the Apple device each time the user presses the tag button. The accessory design must allow tagging while data is being written to the Apple device.

The accessory must create a file for each tag saved when the Apple device is connected and the tag is not ambiguous. The accessory must write ambiguous tags to the same file when the Apple device is connected and the tags are ambiguous. The accessory must write all tags stored locally to one file the next time an Apple device is connected, including all ambiguous tags.

Note: The fields `MajorVersion`, `MinorVersion`, `ManufacturerID`, `ManufacturerName`, `DeviceName`, `MarkedTracks`, `Name`, and `Artist` are required to store a tag file. The `iTunesSongID` field is desirable but not required.

An accessory must never write a tag more than once. If the user presses the tag button multiple times during the same song, the accessory must write the tag to the Apple device after the first button press and then filter out redundant tags by setting a flag to indicate that the broadcast tag data has been written to the Apple device.

The accessory must store tags internally when no Apple device is connected. If the accessory runs out of storage space, it must prompt the user to connect an Apple device. When tags are stored locally, the accessory must write all tag data to a single file the next time an Apple device is docked. The `MajorVersion`, `MinorVersion`, `ManufacturerName`, `ManufacturerID`, and `MarkedTracks` fields need to be written only once per file when writing multiple tags to a single file. The rest of the tag data is written in a tag array with each tag designated by the `<dict>` key. "Sample Tag Files" (page 492) shows an example of multiple tags in a single file.

Before it deletes any tag stored locally, the accessory must verify that the Apple device has sent an `iPodAck` command in response to each `WriteiPodFileData` command and the `CloseiPodFile` command. The accessory must notify the user that the tags were successfully written to the Apple device.

A.3 Accessory User Interface

Table A-4 (page 474) lists the user interface elements that must be used for the tagging feature in a radio accessory. The table shows three sections: Required UI, Optional UI, and UI not allowed. Table A-5 (page 475) lists user interface messages that may appear on the accessory's display.

Table A-4 Tagging feature user interface implementation

Action		Implementation	
		Speaker	**Head unit**
Required UI	Indicate tag available	Button (LED) illuminates	Button or logo/type appears
		Logo/type appears (LCD)	Button color or logo/type changes
	Indicate tag captured	Button LED blinks	Button blinks
		Button LED changes color	Button changes color
		Logo/type blinks (LCD)	Graphic
		Message on LCD	Message on screen
		Audible feedback	Audible feedback
	Indicate accessory memory full	Message (LCD)	Message
		Audible feedback	Audible feedback

Action		Implementation	
		Speaker	Head unit
			Graphic
	Indicate Apple device memory full	Message (LCD)	Message
		Audible feedback	Audible feedback
Optional UI	Indicate write to Apple device	Button LED blinks twice	Button blinks twice
		Message on LCD	Message on screen
		Audible feedback	Audible feedback
	Indicate tag not captured	LED color changes	Button color changes
		Message on LCD	Message on screen
		Audible feedback	Audible feedback
	Tags remaining	Message on LCD	Message on screen
UI not allowed	Resolving tag ambiguity		
	Choosing whether or not to write tags to the Apple device		

Table A-5 Tagging feature UI text messages

Condition	Message
Capturing tags	“Tag available"
	“Tag stored"
	“Tag stored. XX remaining.”
Accessory memory full	“Memory full. Connect iPod.”
	“Connect iPod to transfer tags.”
Apple device memory full	“iPod full. Tags cannot be stored.”
Write to Apple device	“Tags transferred to iPod.”
	“Tags saved to iPod. XX remaining.”

Note: These message texts are preliminary and are provided for guidance only. The specific texts to be displayed are expected to change as the tagging feature user interface is further defined.

The accessory should not have a UI function that deletes individual tags; this is done by iTunes. A function to delete all tags is acceptable as a way to restore the accessory to its factory settings.

A.3.1 Tag Button Text

The button on the radio accessory used to tag a song should bear the word “Tag.” The word “Tag” should be presented in a manner consistent with the text on the accessory’s other buttons (the same font, weight, capitalization, color, illumination, and so on).

A.4 Sample Command Sequence

Table A-6 (page 476) shows a sample sequence of commands that implements radio tagging in an accessory.

Table A-6 Radio tagging command sequence

Step	Accessory command	Apple device command	Comment
1	`StartIDPS`		no params
2		`iPodAck`	acknowledging 'Success (OK)' to command 'General Lingo::StartIDPS'
If the `iPodAck` reply to `StartIDPS` returns a status of 0x00, the accessory proceeds to Step 3. Otherwise, ■ If the accessory receives no `iPodAck` command within 500 ms, it may retry Step 1 at 500 ms intervals as long as Apple Device Detect (pin 30 of the 30-pin connector) is low and Accessory Power (pin 13) is high. Accessories that don’t monitor these signals (such as Bluetooth accessories or USB accessories without a 191 kΩ ID resistor) may retry without checking them. ■ If the accessory receives an `iPodAck` status of 0x04 (Bad Parameter), the Apple device does not support IDPS. Within 800 ms the accessory must send an `IdentifyDeviceLingoes` command with all fields set to 0xFF. See “Canceling a Current Authentication Process With IdentifyDeviceLingoes” (page 135). A sample command sequence is listed in Table C-54 (page 571). ■ If the accessory receives any other nonzero `iPodAck` status, it must not send any more iAP commands until it has been disconnected. Upon reconnection it must restart identification at Step 1.			
3	`RequestTransport-MaxPayloadSize`		requesting transport maximum payload size
4		`ReturnTransport-MaxPayloadSize`	returning transport maximum payload size
5	`GetiPodOptions-ForLingo`		getting options for General Lingo

Step	Accessory command	Apple device command	Comment
6		RetiPodOptions-ForLingo	returning options of 000000000003F3FF (Line out usage | Video output | NTSC video signal format | PAL video signal format | Composite video out connection | S-Video video out connection | Component video out connection | Closed captioning (video) | Video aspect ratio 4:3 (fullscreen) | Video aspect ratio 16:9 (widescreen) | reserved | app communication capable | iPod notifications) for General Lingo
7	GetiPodOptions-ForLingo		getting options for Storage Lingo
8		RetiPodOptions-ForLingo	returning options of 0000000000000003 (iTunes tagging | Nike + iPod cardio equipment) for Storage Lingo
9	SetFIDTokenValues		setting 8 FID tokens ; IdentifyToken = (lingoes: 0/12 | options 0x00000002 | device ID 0x00000200); AccCapsToken = 0x0000000000000805; AccInfoToken = Acc name (Radio); AccInfoToken = Acc FW version (v1.0.1); AccInfoToken = Acc HW version (v1.0.0); AccInfoToken = Acc manufacturer (Radio Manufacturer); AccInfoToken = Acc model number (M78901LL/Z); iPodPreferenceToken = (setting 'used' for preference class 'line out usage' with restore on exit 'selected')
10		AckFIDTokenValues	8 ACKs for FID tokens ; IdentifyToken = accepted; AccCapsToken = accepted; AccInfoToken = (Acc name) accepted; AccInfoToken = (Acc FW version) accepted; AccInfoToken = (Acc HW version) accepted; AccInfoToken = (Acc manufacturer) accepted; AccInfoToken = (Acc model number) accepted; iPodPreferenceToken = (line out usage) accepted
11	EndIDPS		status 'finished with IDPS; proceed to authentication'
12		IDPSStatus	status 'ready for auth'
13		GetAccessory-AuthenticationInfo	no params

Step	Accessory command	Apple device command	Comment
14	RetAccessory-AuthenticationInfo		returning auth protocol v2.0; current section index: 0; maximum section index: 1; cert data ...
15		iPodAck	acknowledging 'Success (OK)' to command 'General Lingo::RetAccessoryAuthenticationInfo'
16	RetAccessory-AuthenticationInfo		returning auth protocol v2.0; current section index: 1; maximum section index: 1; cert data ...
17		AckAccessory-AuthenticationInfo	acknowledging 'auth info supported'
18		GetAccessory-Authentication-Signature	offering challenge ...
19	RetAccessory-Authentication-Signature		returning signature ...
20		AckAccessory-AuthenticationStatus	acknowledging authentication status 'Success (OK)'
21	GetiPodCaps		no params
22		RetiPodCaps	returning 'total space 2147483648; max file size 1073741824; max write size 500; v1.02'
23	GetiPodFreeSpace		no params
24		RetiPodFreeSpace	returning 130023424 bytes
25	OpeniPodFeatureFile		opening feature type 'Radio Tagging'
26		RetiPodFileHandle	returning file handle 0
27	WriteiPodFileData		writing 184 bytes at offset 0 and handle 0: <?xml version=""1.0" encoding=""UTF-8""?> <!DOCTYPE plist PUBLIC ""-//Apple Inc//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd""> <plist version=""1.0""> <dict>
28		iPodAck	acknowledging 'Success' to command 'Storage Lingo::WriteiPodFileData' with file handle 0

Step	Accessory command	Apple device command	Comment
29	WriteiPodFileData		writing 290 bytes at offset 184 and handle 0: <key>DeviceName</key> <string>IES2</string> <key>MajorVersion</key> <integer>1</integer> <key>MinorVersion</key> <integer>0</integer> <key>ManufacturerID</key> <integer>17</integer> <key>ManufacturerName</key> <string>Polk Audio</string> <key>MarkedTracks</key> <array> <dict>
30		iPodAck	acknowledging 'Success' to command 'Storage Lingo::WriteiPodFileData' with file handle 0
31	WriteiPodFileData		writing 472 bytes at offset 474 and handle 0: <key>Name</key> <string>The Rising</string> <key>Artist</key> <string>Bruce Springsteen</string> <key>Album</key> <string>The Rising</string> <key>StationURL</key> <string>http://www.hdradio.com </string> <key>iTunesSongID</key> <integer>192903160</integer> <key>StationCallLetters</key> <string>IHDR</string> <key>StationFrequency</key> <string>88.1 FM</string> <key>StreamID</key> <string>2</string> <key>ProgramType</key> <string>Classic Rock</string>
32		iPodAck	acknowledging 'Success' to command 'Storage Lingo::WriteiPodFileData' with file handle 0
33	WriteiPodFileData		writing 210 bytes at offset 946 and handle 0: <key>TimeStamp</key> <date>2009-06-01T09:45:57Z </date> <key>iTunesStorefrontID</key> <integer>143441</integer> <key>iTunesPlaylistID</key> <integer>192901525</integer> </dict> </array> </dict> </plist>

Step	Accessory command	Apple device command	Comment
34		`iPodAck`	acknowledging 'Success' to command 'Storage Lingo::WriteiPodFileData' with file handle 0
35	`CloseiPodFile`		closing file with handle 0
36		`iPodAck`	acknowledging 'Success' to command 'Storage Lingo::CloseiPodFile' with file handle 0

A.5 FM Radio Tagging

FM radio uses the RDS/RBDS subcarrier system to broadcast data, in addition to its audio content. For information about RBDS, see the *United States RBDS Standard, NRSC-4-A*, available at www.nrscstandards.org. FM tagging data is broadcast inside two ODAs (Open Data Applications): a RadioText Plus (RT+) ODA and a private iTunes Tagging ODA.

The FM radio accessory must capture the data streams from both these ODAs and write their information to plist files, as described in "Capturing and Storing Tag Data" (page 467). During this process, the accessory must populate every plist field listed in Table A-7 (page 480) for which it receives metadata from the radio broadcast. For an FM tag to be usable, its plist entry must provide valid data for at least the `Title` and `Artist` fields.

Table A-7 Required plist fields for FM

Name	Type	Description	FM Source
Name	string	The name of the track.	RT+ type: ITEM.TITLE
Artist	string	The name of the artist.	RT+ type: ITEM.ARTIST
Album	string	The name of the album.	RT+ type: ITEM.ALBUM
Genre	string	The genre classification of the track.	RT+ type: ITEM.GENRE
iTunesSongID	integer	The iTunes song identifier.	iTunes ODA: iTunesSongID
iTunesStorefrontID	integer	The iTunes storefront identifier.	iTunes ODA: iTunesStorefront
StationFrequency	string	The station's frequency, expressed only by digits and an optional decimal point.	Accessory
StationCallLetters	string	Station call letters can be extracted from the RDS PI field or through the RT+ STATIONNAME.SHORT and STATIONNAME.LONG classes.	RT+ type: STATIONNAME.SHORT; STATIONNAME.LONG

Name	Type	Description	FM Source
TimeStamp	date	The date and time at which the tagged track was broadcast, in ASCII timestamp format (e.g., 2008-04-16T00:35:42Z). The time zone must be UTC; iTunes converts it into local time.	RDS: Group 4A
PodcastFeedURL	integer	Indication that the broadcast has an associated podcast. If RT+ type INFO.URL is filled, the broadcast should be marked as having an associated podcast. Set to 1 if there is a Podcast URL stored in the StationURL field.	N/A
StationURL	string	The URL of the broadcast station as taken from the RT+ type PROGRAMME.HOMEPAGE, or of a podcast if PodcastFeedURL is 1 as taken from the RT+ type INFO.URL.	RT+ type: PROGRAMME.HOMEPAGE; INFO.URL
ProgramNumber	integer	The station's broadcast subchannel.	RT+ type: P PROGRAMME.SUBCHANNEL
iTunesStationID	string	The iTunes station ID, as broadcast. The lower byte of the PI code concatenated with the Region ID code.	RDS: PI code and Region ID code
UnknownData	data	A field used to pass unknown iTunes Tagging ODA data types to iTunes. See "Handling Unknown Metadata Types" (page 469).	iTunes ODA

A.5.1 The RT+ ODA

RadioText Plus (RT+) is a technology that tags RDS RadioText (RT) messages so specific content can be retrieved from them. RT+ tags are transmitted in RDS broadcasts as an ODA with an Application ID (AID) of 0x4BD7. You can obtain the RT+ specification from the RDS Forum at www.rds.org.uk/.

Each RT+ tag contains three elements:

- The RT content type
- The position inside the RT message of the first character of content to be retrieved
- The length of the content to be retrieved.

The RT+ content types used for iTunes FM radio tagging are listed in Table A-8 (page 481). The corresponding plist fields sent to the Apple device are described in Table A-3 (page 471).

Table A-8 RT+ content types

RT+ type	iTunes plist field
`ITEM.TITLE`	`Name`
`ITEM.ARTIST`	`Artist`

RT+ type	iTunes plist field
ITEM.ALBUM	Album
ITEM.GENRE	Genre
STATIONNAME.SHORT, STATIONNAME.LONG	StationCallLetters
INFO.URL	PodcastFeedURL
PROGRAMME.HOMEPAGE	StationURL
PROGRAMME.SUBCHANNEL	ProgramNumber

A.5.2 The iTunes Tagging ODA

The iTunes ODA used for FM broadcasts encodes tag information as **elements.** Currently 6 element types are defined and another 10 are reserved for future use, as shown in Table A-9 (page 482).

Table A-9 FM tag element types

ID type	Description
0x0	Event Control
0x1	iTunes Song ID
0x2	iTunes Artist ID
0x3	Reserved
0x4	Industry ID
0x5	iTunes Storefront
0x6	Extended ID
0x7-0xF	Reserved

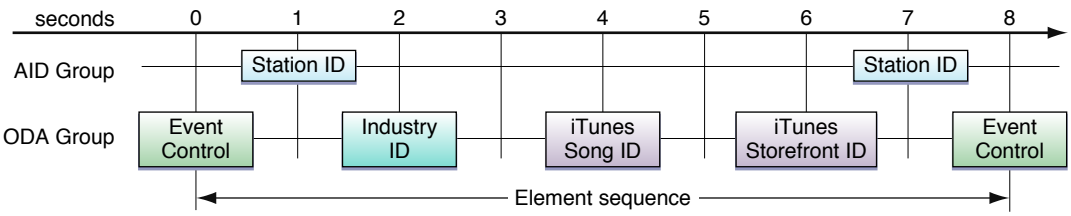
All currently defined FM tag elements carry 32 bits of data and are transmitted using a single ODA application group. If necessary, future versions of the iTunes FM tagging protocol may accommodate larger data types by chaining several ODA groups together.

FM tag elements are normally encrypted before transmission. Because the transmission must be decrypted by the receiver, it is keyed to a composite of values contained in the RDS data. This process is described in "FM Tag Decryption" (page 486).

A complete iTunes tagging transmission normally consists of several element types in an **element sequence.** The elements in the sequence are transmitted at a relatively low rate (about one element every 2 seconds), so the RDS bandwidth consumed by iTunes tagging is relatively small. When a complete sequence of elements has been transmitted the process repeats itself. Figure A-1 (page 483) shows the transmission timeline of a typical element sequence. In this example, the sequence consists of 4 element types and a complete transmission of the sequence occurs every 8 seconds.

Note: To assure that tag data is always available, the maximum recommended time between transmissions is 10 seconds.

Figure A-1 iTunes tagging element sequence



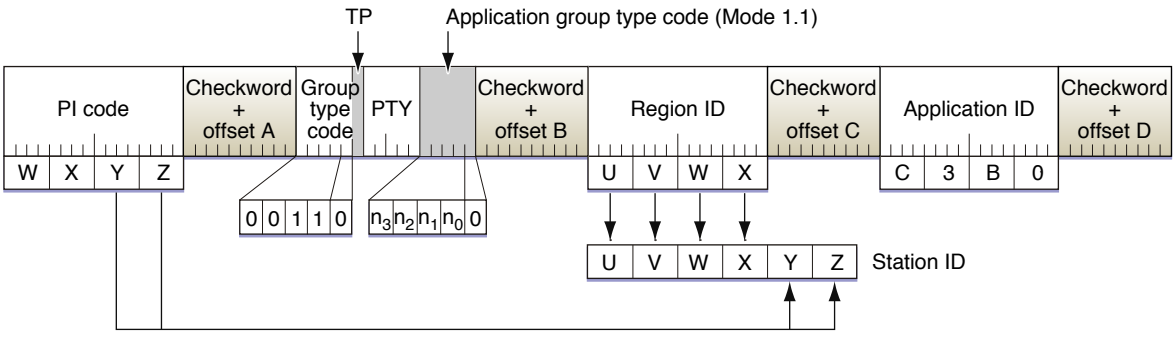
A.5.3 Use of the RDS Group Type 3A

Because the RT+ and FM iTunes tagging data are contained in ODAs, they use the RDS group type 3A, also called the AID (Application ID) Group type. This RDS group type includes 16 message bits for use by each ODA (block C) and indicates the application group type (if any) used for transmission of additional ODA data. Tag data is transmitted in mode 1.1, meaning that the broadcaster allocates an available type A group for transmission of iTunes tag element data. RDS 3A group messages with Application ID 0xC3B0 indicate which RDS group type will be used to carry the iTunes tagging information. The accessory must parse group type 3A messages and save the application group type code (n3-n0), because this indicates the ODA group type on which tag data will be broadcast.

Note: The accessory must parse and save the application group type code for each iTunes-specific type 3A message, because the type code for subsequent iTunes data may change during the broadcast.

The recommended transmission rate for the iTunes tag 3A group is once every 7 seconds, with a maximum repeat delay of once every 10 seconds. Upon tuning to a new frequency, a tagging-enabled FM receiver waits for the reception of a 3A group containing the assigned Application ID number; this indicates that the broadcast includes iTunes tag data. The RDS 3A group structure for iTunes tagging is shown in Figure A-2 (page 483).

Figure A-2 RDS 3A group for iTunes tagging



The Region ID code shown in Figure A-2 (page 483) combines the extended country code (see Annex N of the RBDS specification) and the upper byte of the PI (Program Identification) code: the upper byte "UV" of the Region ID carries the extended country code and the lower byte "WX" mirrors the upper byte of the PI code. The receiver must combine the 16-bit Region ID with the lower byte of the PI code to form a 24-bit Station ID, a guaranteed unique identifier of a particular radio station. The 24-bit Station ID has the byte order UVWXYZ, as shown in the diagram.

Note: It is possible that the upper byte of the PI code may not exactly mirror the lower byte of the Region ID. For this reason, a receiver must construct the Station ID exactly as specified above. Always use the bytes transmitted in Region ID; ignore the upper byte of the PI code.

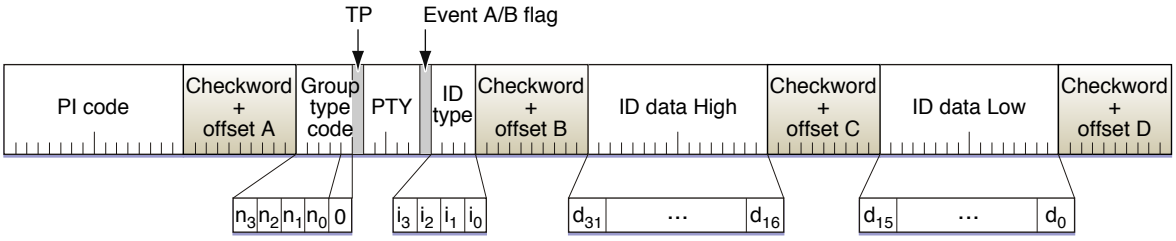
The Station ID is intended for use in affiliate programs. Because affiliate programs normally involve an online merchant, that merchant or an appropriate third party maintains a table translating the Station ID into a more traditional Affiliate ID. For example, a range of Station ID codes may map to a single group identifier for the purposes of an affiliate program.

A.5.4 Tagging Application Group

The iTunes tag element data is transmitted using an RDS type A application group (also called an ODA Group), as shown in Figure A-3 (page 484). The Group Type Code (n3-n0) of this message should correspond to that broadcast as the application group type code, which was part of the Group 3A message with AID 0xC3B0.

Figure A-3 (page 484) depicts the data before encryption at the transmitter and following successful decryption at the receiver. The encryption process scrambles the ID Type (bits i3 to i0) as well as the ID data (bits d31 to d16 and d15 to d0). The remaining data (PI code, group type code, TP flag, PTY and Event A/B flag) are always transmitted in the clear.

Figure A-3 RDS application group for iTunes tagging



The ID Type indicates the specific type of element data carried by an instance of the ODA group. There are 16 possible element types of which 6 are currently defined (see Table A-9 (page 482)). The unassigned elements are available for future use. Types 00000 and 1111 are reserved as special-purpose control elements; all other types are available for content identification codes. Data received with an unassigned element type must be written to an `UnknownData` field in the plist sent to the Apple device; see "Handling Unknown Metadata Types" (page 469).

The ID data field carries the actual data associated with an element. For all currently-defined numeric identifiers (iTunes Song, iTunes Artist, iTunes Storefront, Extended ID and Industry ID) these bits are interpreted as 32-bit unsigned integer values. The Event Control element uses an alternate definition of the data bits, as detailed in "FM Tagging Event Control" (page 485). The Event A/B flag is a toggle bit that changes state each time the broadcast content changes. This tells the receiver that it should clear any temporary buffers associated with element data in preparation for reception of a new set of data. In a typical music broadcast, the Event A/B

flag toggles at the transition from one song to the next, at the transition from music to non-music content (advertisement break) and at the transition from non-music content back to music. Even if there are no elements associated with a particular piece of content, the A/B flag still toggles at the start of that content to mark the end of previously identified content. In such a situation, an element sequence consisting only of the Event Control element is transmitted as a carrier for the A/B flag.

A.5.5 Normal Mode and Test Mode

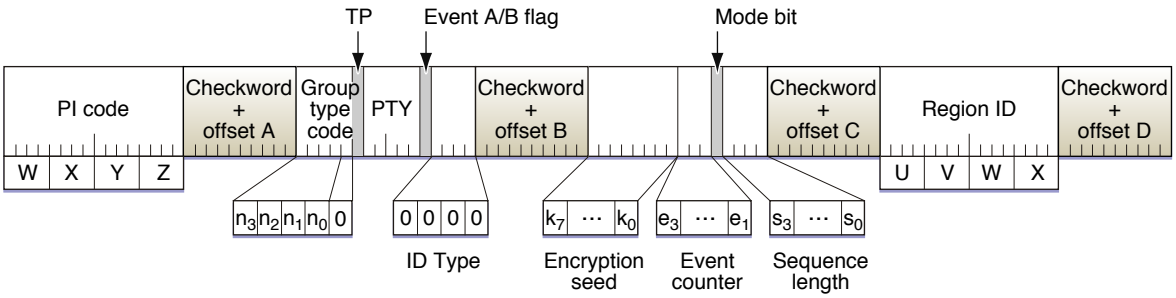
Two modes of operation are defined for the RDS transmission of iTunes tag data: **normal mode** and **test mode.** The Region ID section of Group Type Code 3A messages with AID 0xC3B0 indicate what mode is being used for broadcast. In normal mode the message bits of the AID group carry a 16-bit Region ID and encryption/decryption is enabled. In test mode the message bits are set to zero (0x0000) and encryption/decryption is disabled. Tagging-enabled FM receivers must support both operating modes. In test mode, the Station ID is constructed using Region ID and the PI code lower byte as carried by the Event Control element (See "FM Tagging Event Control" (page 485)). In test mode it is not possible to construct a Station ID from the 3A group because the Region ID is set to zero.

A.5.6 FM Tagging Event Control

Event Control is a special-purpose element used to indicate a change in the content being broadcast as well as to indicate the overall status of the iTunes tag transmission.

The **event counter** is a 4-bit count that is incremented each time the on-air content changes. After the count reaches 15 it wraps back to 0. The 3 most significant bits of the count (bits e3-e1) are carried in the Event Control element, shown in Figure A-4 (page 485). The least significant bit of the counter (bit e0) is carried in every element type and is called the Event A/B Flag.

Figure A-4 Event Control element



When set to 1, the mode bit shown in Figure A-4 (page 485) indicates that the event currently being broadcast has been successfully mapped to at least one type of content identifier and that identifiers are being transmitted as part of the element sequence. If no mapping exists for the current event, the mode bit is set to 0. The Sequence length bits (bits s3-s0) indicate the number of element types included in the element sequence for the current event. A value of 0000 indicates that the sequence consists only of the Event Control element. The decryption seed (bits k7-k0) is a one-byte value used to seed the decryption system discussed in "FM Tag Decryption" (page 486). The 16-bit Region ID value transmitted in block C of the AID group is also transmitted as block D of the Event Control element.

Note: The decryption seed must be read from every Event Control message because it may change during the broadcast.

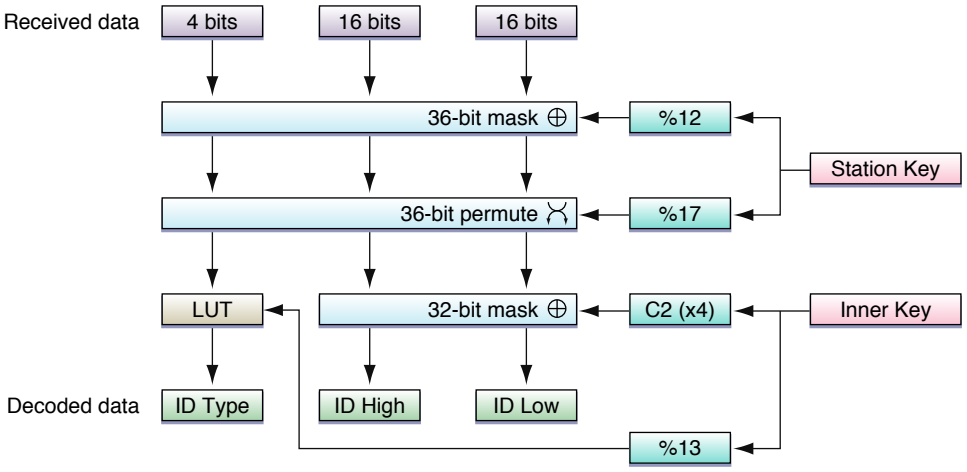
In normal mode, this retransmission lets a receiver quickly verify that Region ID has been received error-free and that element data is being decoded correctly. Retransmission is particularly valuable because the Region ID (as a component of the Station ID) is used as part of the decryption process. In test mode, the Region ID is transmitted only as part of the Event Control element. Retransmission is of less value in test mode because encryption/decryption is disabled.

A.5.7 FM Tag Decryption

During the transmission of FM tag elements, 36 bits of raw data are passed into an encryption process, including the 4-bit element type ID and 32 bits of element data (ID High and ID Low). The encryption algorithm is keyed by two values generated from RDS data: the Station Key and the Inner Key. The Station Key is the algebraic sum of all three bytes of the Station ID: `StationKey = YZ + UV + WX`. The Inner Key is the linear combination (XOR) of three byte values: the 8 least significant bits of the Station Key, a byte formed from Sequence Length (upper nibble) and Event Counter (lower nibble), and the one-byte Encryption Seed carried by the Event Control element: `InnerKey = (StationKey & 0xFF) ^ ((SequenceLength << 4) | EventCounter) ^ EncryptionSeed`.

The decryption process for FM iTunes tagging elements is diagrammed in Figure A-5 (page 486) and is described below.

Figure A-5 Element decryption process



To decode the received data, it is first necessary to generate the station key, which is the algebraic sum of all three bytes of the Station ID. (`StationKey = YZ + UV + WX`). The three input bytes and the result are treated as unsigned integers. The Station ID bytes can be recovered from the AID group.

Upon reception of an ODA group, the 36 bits of coded tag data are passed into the decryption process. The first stage of decoding applies 1 of 12 predefined outer masks to the full 36 bits of received data using a linear mixing (XOR) operation. The 12 predefined masks are listed in Table A-10 (page 487). The mask used is selected as the modulus (integer remainder) of the Station Key divided by 12 (`MaskRow = StationKey % 12`).

Table A-10 Outer mask lookup table

Row	Bits 35-32	Bits 31-16	Bits 15-0
0	D	D5D0	E3E1
1	2	6073	04B4
2	C	5C59	80D5
3	E	9C6E	78D0
4	3	9E48	2754
5	4	062E	9DB7
6	B	2924	4436
7	6	F308	3628
8	9	9DE8	27A4
9	5	1C71	CEA0
10	8	B5C7	C134
11	7	679C	30BD

The second stage of decoding applies 1 of 17 predefined permute operations to the full 36 bits of received data. The permutation operation reorders the data at the bit level. The 17 permutation operations are shown in Table A-11 (page 487) and Table A-12 (page 488). The operation used is selected as the modulus of the station key divided by 17 (`PermuteRow = StationKey % 17`).

Table A-11 Permutation lookup table, upper bits

Bits	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18
0	7	11	4	35	3	5	26	2	21	32	6	20	17	0	14	27	9	15
1	13	29	28	12	19	33	32	15	24	27	2	9	4	34	17	8	16	22
2	25	9	24	13	18	11	5	32	22	17	35	16	2	29	6	15	1	19
3	16	20	2	0	28	14	6	25	12	8	7	1	32	24	4	19	33	9
4	10	8	20	7	16	28	33	21	35	1	15	12	25	17	11	22	18	23
5	15	17	16	19	7	3	30	8	28	12	29	34	11	18	25	13	2	31
6	21	31	11	6	34	13	0	33	17	9	26	10	14	7	2	18	22	20
7	4	22	9	15	14	12	1	6	7	21	33	19	35	11	16	29	10	5
8	0	4	35	18	6	7	24	23	11	34	32	3	1	31	22	16	8	30

Bits	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18
9	18	2	15	20	5	22	3	19	4	25	31	26	0	28	13	7	29	1
10	3	0	5	9	17	1	8	18	13	31	4	15	6	12	20	14	23	25
11	9	6	1	33	30	32	21	29	19	13	0	22	10	5	7	2	14	12
12	28	24	26	34	33	18	14	5	25	15	10	0	9	32	19	31	17	13
13	31	5	13	14	0	35	17	12	30	29	22	18	24	25	1	21	4	26
14	22	32	23	24	15	16	35	11	6	33	18	14	28	21	29	0	27	34
15	2	30	0	4	13	15	18	9	8	20	28	32	21	10	31	6	11	35
16	19	33	10	2	27	17	22	7	16	3	21	5	30	20	24	35	32	14

Table A-12 Permutation lookup table, lower bits

Bits	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	8	23	29	30	22	19	16	25	24	10	1	33	12	28	34	18	13	31
1	18	10	31	11	26	3	35	6	25	23	5	0	1	30	7	20	14	21
2	10	8	30	27	23	7	31	12	33	3	0	34	20	21	26	14	28	4
3	5	3	34	31	35	30	21	26	22	15	13	29	18	10	11	23	17	27
4	13	14	2	24	30	29	34	19	26	31	32	4	3	27	9	0	5	6
5	20	6	24	26	10	1	0	23	27	21	4	22	14	5	35	33	32	9
6	1	24	28	4	32	5	15	8	23	29	12	25	30	3	27	19	16	35
7	32	27	20	3	34	31	18	13	30	0	23	28	24	2	25	17	26	8
8	21	2	26	29	12	9	20	17	13	33	25	5	10	14	15	28	27	19
9	27	11	16	8	6	17	24	32	35	9	34	30	21	23	14	12	10	33
10	33	19	27	7	2	21	30	28	34	24	10	16	26	22	29	35	11	32
11	34	31	11	23	20	25	8	27	3	4	15	17	28	26	18	16	35	24
12	35	4	21	6	7	23	29	20	8	30	2	3	27	1	16	11	22	12
13	2	33	7	19	16	20	28	3	32	6	8	27	23	34	10	15	9	11
14	3	26	19	1	31	2	4	5	20	17	30	13	9	12	8	10	7	25
15	26	17	5	12	29	16	27	24	19	34	22	23	25	33	1	7	3	14
16	23	1	8	25	4	26	13	15	31	18	29	12	6	11	0	9	34	28

Note: At this stage the outer layer of the encryption system has been reversed and it is now possible to read data from the control elements (upper 4 bits set to 0000 or 1111). The reception and parsing of an event control element (type 0000) is required before the remaining ID oriented elements can be decoded.

The third stage decodes the ID type by passing the upper 4 data bits through a predefined lookup table. It then decodes the ID high and ID low bits by applying an algorithmically-generated inner mask to the lower 32 data bits. The ID lookup is shown in Table A-13 (page 489). The lookup used is selected as the modulus of the inner key divided by 13 (`IDlookupRow = InnerKey % 13`). The upper 4 data bits select the table column and the decoded ID Type is the cell value at the selected row and column.

Table A-13 Element type lookup table

ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	6	7	2	9	8	11	4	3	10	13	14	1	12	5	15
1	0	3	10	4	5	6	7	11	2	12	9	1	13	14	8	15
2	0	9	3	11	6	7	4	1	14	13	12	5	8	2	10	15
3	0	14	1	9	13	4	12	6	10	11	3	8	2	5	7	15
4	0	7	14	8	3	10	2	12	13	1	5	4	11	9	6	15
5	0	13	11	12	1	14	5	8	4	3	6	10	9	7	2	15
6	0	11	4	13	14	2	9	3	7	8	1	6	5	10	12	15
7	0	2	9	7	10	12	14	13	5	4	8	3	6	1	11	15
8	0	8	13	6	2	9	10	5	11	14	4	12	7	3	1	15
9	0	4	12	1	11	13	8	10	9	5	2	7	14	6	3	15
10	0	5	8	14	7	1	3	9	12	6	11	2	10	4	13	15
11	0	10	6	5	12	11	13	2	1	7	14	9	3	8	4	15
12	0	12	5	10	8	3	1	14	6	2	7	13	4	11	9	15

Note: At this point the inner layer of the encryption process for the element type has been reversed and all element types are fully decoded. Decryption of the data bits is accomplished by passing them through the polynomial shown in Listing A-1 (page 490).

To decode ID-oriented elements, it is necessary to generate the inner key. The inner key is a linear mixing (XOR) of three byte values: the 8 least significant bits of the station key, a byte formed from the sequence lLength (upper nibble) and event counter (lower nibble) and the encryption seed value, all carried by the event control element (`InnerKey = (StationKey & 0xFF) ^ ((SequenceLength << 4) | EventCounter) ^ EncryptionSeed`).

The inner mask is a pseudo-random bit pattern output from a linear feedback shift register seeded by the Inner Key. The Baicheva C2 algorithm is used as the generator polynomial. Four calls are made to the random number generator, each of which produce 8 bits of data. The seed value of the first call is the inner key. The result of the first call becomes the upper 8 bits of the inner mask. The result of the first call also becomes the seed value of the second call, and so on. The result of the fourth call becomes the lower 8 bits of the inner mask.

A code listing for the pseudo-random number generator is shown in Listing A-1 (page 490). To build the 32-bit inner mask, four calls are made to the `IDI_C2` function. The initial call passes in the seed value as the parameter `M`. Passing the output of a call (the return value) in as the seed value of a subsequent call produces the required pseudo-random sequence. Each call produces 8 of the 32 inner mask bits.

Listing A-1 Pseudo-random number generator

```
int IDI_C2( int M )
{
 int S = 0x0000;

 for ( int i = 0x0080; i > 0; i >>= 1 )
 {
  if ( (i & M) != 0 )
  {
   S = ((( S << 1 ) ^ 0x0100 ) & 0x01FF );
  }
  else
  {
   S = (( S << 1 ) & 0x01FF );
  }

  if ( (S & 0x0100) != 0 )
  {
   S = (( S ^ 0x012F ) & 0x01FF );
  }
 }

 return S;
}
```

A reference implementation of the decryption process, in generic source code, is available from Apple upon request.

A.6 HD Radio Tagging

In addition to their audio content, HD radio broadcasts contain Program Service Data (PSD) and Station Information Service (SIS) data. The HD radio accessory must capture these data streams and write their information to the plist fields listed in Table A-14 (page 491). The accessory must populate every plist field for which it receives data from the radio broadcast, including program service data (PSD) and station information services (SIS).

Table A-14 Required plist fields for HD radio

Name	Type	Description	HD Radio Source
Name	string	The name of the track.	HD PSD
Artist	string	The name of the artist.	HD PSD
Album	string	The name of the album.	HD PSD
Genre	string	The genre classification of the track as communicated through the HD program type. The receiver must write this field as received, without decoding. It can contain ID3 codes, 2-digit genre strings, or both.	HD PSD
iTunesSongID	integer	The iTunes song identifier.	HD PSD
iTunesStorefrontID	integer	The iTunes storefront identifier.	HD PSD
StationFrequency	string	The station's frequency, expressed only by digits and an optional decimal point.	HD SIS
StationCallLetters	string	The call letters for the station, written exactly as received. Call letters may be up to 12 characters long if the Universal Short Station Name is broadcast.	HD SIS
TimeStamp	date	The date and time at which the tagged track was broadcast, in ASCII timestamp format (e.g., 2008-04-16T00:35:42Z). The time zone must be UTC; iTunes converts it into local time. The TimeStamp plist field should be written only if the receiver has received valid, GPS-locked time information over the air and has created a valid timestamp based on it. Receivers using TI DRI352 baseband processor firmware prior to version 17 or NXP SAF355X baseband processor firmware prior to Version 2.1.2.1 should never write this field, because only newer baseband processor versions process the timestamp information correctly. Contact iBiquity for further details.	HD SIS
PodcastFeedURL	integer	Indication that the broadcast has an associated podcast. Set to 1 if there is a Podcast URL stored in the StationURL field.	HD PSD
StationURL	string	The URL of the broadcast station.	HD PSD
ProgramNumber	integer	The station's multicast channel in the range 1 to 8, where 1 indicates the main program.	HD HOST
iTunesAffiliateID	string	The iTunes affiliate ID, as broadcast.	HD PSD
UnknownData	data	A field used to pass unknown UFID data to iTunes. See "Handling Unknown Metadata Types" (page 469).	HD PSD (UFID data)

A.7 Satellite Radio Tagging

In addition to their audio content, Satellite radio broadcasts contain Service Element data. The Satellite radio accessory must capture these data streams and write their information to the plist fields listed in Table A-15 (page 492). The accessory must populate every plist field for which it receives data from the radio broadcast.

Table A-15 Required plist fields for Satellite radio

Name	Type	Description	Satellite Radio Source
Name	string	The name of the track.	Sirius/XM Service Element: TrackName
Artist	string	The name of the artist.	Sirius/XM Service Element: ArtistName/ArtistLabel
iTunesSongID	integer	The iTunes song identifier.	Sirius/XM Service Element: iTunesSongID
StationFrequency	string	The station's channel number.	Sirius/XM Service Element: ChannelNumber
StationCallLetters	string	The name of the broadcast station.	Sirius/XM Service Element: ChannelName
StationURL	string	The URL of the broadcast station's Web site.	Sirius/XM Service Element: URL
TimeStamp	date	The date and time at which the tagged track was broadcast; iTunes converts it into local time.	Sirius/XM Service Element: Date/Time
iTunesAffiliateID	string	The iTunes affiliate ID, as broadcast.	As assigned to Satellite Radio Receiver

A.8 Other Broadcast Tagging

In addition to their audio content, some broadcasts contain metadata that describes the currently playing broadcast audio. In these cases the radio accessory may capture these data streams and write their information to the plist fields listed in Table A-3 (page 471). The accessory must populate every plist field for which it receives data from the broadcast.

A.9 Sample Tag Files

This section lists sample plist files that might be generated by FM, HD, and satellite radio tagging.

A.9.1 FM Radio Sample

The following is a sample FM radio plist-formatted tag file showing a header and one stored tag:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Inc//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
   <key>MajorVersion</key>
   <integer>1</integer>
   <key>MinorVersion</key>
   <integer>1</integer>
   <key>ManufacturerID</key>
   <integer>17</integer>
   <key>ManufacturerName</key>
   <string>Acme</string>
   <key>DeviceName</key>
   <string>AC-HDR100</string>
   <key>MarkedTracks</key>
   <array>
       <dict>
           <key>Name</key>
           <string>Times Like These</string>
           <key>Artist</key>
           <string>Foo Fighters</string>
           <key>Album</key>
           <string>One by One</string>
           <key>iTunesSongID</key>
           <integer>6906304</integer>
           <key>iTunesStorefrontID</key>
           <integer>143441</integer>
           <key>StationFrequency</key>
           <string>104.9</string>
           <key>StationCallLetters</key>
           <string>KCNL</string>
           <key>StationURL</key>
           <string>http://www.channel1049.com</string>
           <key>Genre</key>
           <string>(20) Alternative</string>
           <key>TimeStamp</key>
           <date>2007-04-16T00:35:42Z</date>
           <key>ProgramNumber</key>
           <integer>1</integer>
           <key>iTunesStationID</key>
           <string>10491557</string>
           <key>UnknownData</key>
           <data>MDcwNDEyMzQwMTAyMTEwNDA4MTJhYjU2Z2g=</data>
       </dict>
   </array>
</dict>
</plist>
```

A.9.2 HD Radio Samples

The following is a sample HD radio plist-formatted tag file showing a header and one stored tag:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Inc//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>MajorVersion</key>
    <integer>1</integer>
    <key>MinorVersion</key>
    <integer>1</integer>
    <key>ManufacturerID</key>
    <integer>17</integer>
    <key>ManufacturerName</key>
    <string>Acme</string>
    <key>DeviceName</key>
    <string>AC-HDR100</string>
    <key>MarkedTracks</key>
    <array>
        <dict>
            <key>Name</key>
            <string>Times Like These</string>
            <key>Artist</key>
            <string>Foo Fighters</string>
            <key>Album</key>
            <string>One by One</string>
            <key>iTunesSongID</key>
            <integer>6906304</integer>
            <key>iTunesStorefrontID</key>
            <integer>143441</integer>
            <key>StationFrequency</key>
            <string>104.9</string>
            <key>StationCallLetters</key>
            <string>KCNL</string>
            <key>PodcastFeedURL</key>
            <integer>0</integer>
            <key>StationURL</key>
            <string>http://www.channel1049.com</string>
            <key>Genre</key>
            <string>(20) Alternative</string>
            <key>TimeStamp</key>
            <date>2007-04-16T00:35:42Z</date>
            <key>ProgramNumber</key>
            <integer>1</integer>
            <key>iTunesAffiliateID</key>
            <string>00-12uR34oIcpQ</string>
            <key>UnknownData</key>
            <data>MDcwNDEyMzQwMTAyMTEwNDA4MTJhYjU2Z2g=</data>
        </dict>
    </array>
</dict>
</plist>
```

The following is a sample HD radio plist-formatted tag file showing how ambiguous tags are saved to an Apple device:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Inc//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
```

```
<dict>
    <key>MajorVersion</key>
    <integer>1</integer>
    <key>MinorVersion</key>
    <integer>1</integer>
    <key>ManufacturerID</key>
    <integer>17</integer>
    <key>ManufacturerName</key>
    <string>Acme</string>
    <key>DeviceName</key>
    <string>AC-HDR100</string>
    <key>MarkedTracks</key>
    <array>
        <dict>
            <key>AmbiguousTag</key>
            <integer>1</integer>
            <key>ButtonPressed</key>
            <integer>1</integer>
            <key>Name</key>
            <string>Times Like These</string>
            <key>Artist</key>
            <string>Foo Fighters</string>
            <key>Album</key>
            <string>One by One</string>
            <key>iTunesSongID</key>
            <integer>6906304</integer>
            <key>iTunesStorefrontID</key>
            <integer>143441</integer>
            <key>StationFrequency</key>
            <string>104.9</string>
            <key>StationCallLetters</key>
            <string>KCNL</string>
            <key>PodcastFeedURL</key>
            <integer>0</integer>
            <key>StationURL</key>
            <string>http://www.channel1049.com</string>
            <key>Genre</key>
            <string>(20) Alternative</string>
            <key>TimeStamp</key>
            <date>2007-04-16T00:35:42Z</date>
            <key>ProgramNumber</key>
            <integer>1</integer>
            <key>iTunesAffiliateID</key>
            <string>00-12uR34oIcpQ</string>
        </dict>
        <dict>
            <key>AmbiguousTag</key>
            <integer>1</integer>
            <key>ButtonPressed</key>
            <integer>0</integer>
            <key>Name</key>
            <string>Vertigo</string>
            <key>Artist</key>
            <string>U2</string>
            <key>Album</key>
            <string>How to Dismantle an Atomic Bomb</string>
            <key>iTunesSongID</key>
            <integer>29600235</integer>
```

```
            <key>iTunesStorefrontID</key>
            <integer>143441</integer>
            <key>StationFrequency</key>
            <string>104.9</string>
            <key>StationCallLetters</key>
            <string>KCNL</string>
            <key>PodcastFeedURL</key>
            <integer>0</integer>
            <key>StationURL</key>
            <string>http://www.channel1049.com</string>
            <key>Genre</key>
            <string>(17) Rock</string>
            <key>TimeStamp</key>
            <date>2007-04-16T00:35:45Z</date>
            <key>ProgramNumber</key>
            <integer>1</integer>
            <key>iTunesAffiliateID</key>
            <string>00-12uR34oIcpQ</string>
        </dict>
    </array>
</dict>
</plist>
```

The following is a sample HD radio plist-formatted tag file showing how an accessory that has multiple tags stored locally writes those tags to an Apple device:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Inc//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>MajorVersion</key>
    <integer>1</integer>
    <key>MinorVersion</key>
    <integer>1</integer>
    <key>ManufacturerID</key>
    <integer>17</integer>
    <key>ManufacturerName</key>
    <string>Acme</string>
    <key>DeviceName</key>
    <string>AC-HDR100</string>
    <key>MarkedTracks</key>
    <array>
        <dict>
            <key>Name</key>
            <string>Times Like These</string>
            <key>Artist</key>
            <string>Foo Fighters</string>
            <key>Album</key>
            <string>One by One</string>
            <key>iTunesSongID</key>
            <integer>6906304</integer>
            <key>iTunesStorefrontID</key>
            <integer>143441</integer>
            <key>StationFrequency</key>
            <string>104.9</string>
            <key>StationCallLetters</key>
            <string>KCNL</string>
```

```
        <key>PodcastFeedURL</key>
        <integer>0</integer>
        <key>StationURL</key>
        <string>http://www.channel1049.com</string>
        <key>Genre</key>
        <string>(20) Alternative</string>
        <key>TimeStamp</key>
        <date>2007-04-16T00:35:42Z</date>
        <key>ProgramNumber</key>
        <integer>1</integer>
        <key>iTunesAffiliateID</key>
        <string>00-12uR34oIcpQ</string>
    </dict>
    <dict>
        <key>Name</key>
        <string>Vertigo</string>
        <key>Artist</key>
        <string>U2</string>
        <key>Album</key>
        <string>How to Dismantle an Atomic Bomb</string>
        <key>iTunesSongID</key>
        <integer>29600235</integer>
        <key>iTunesStorefrontID</key>
        <integer>143441</integer>
        <key>StationFrequency</key>
        <string>104.9</string>
        <key>StationCallLetters</key>
        <string>KCNL</string>
        <key>PodcastFeedURL</key>
        <integer>0</integer>
        <key>StationURL</key>
        <string>http://www.channel1049.com</string>
        <key>Genre</key>
        <string>(17) Rock</string>
        <key>TimeStamp</key>
        <date>2007-04-16T00:43:45Z</date>
        <integer>1</integer>
        <key>ProgramNumber</key>
        <integer>1</integer>
        <key>iTunesAffiliateID</key>
        <string>00-12uR34oIcpQ</string>
    </dict>
    <dict>
        <key>Name</key>
        <string>Ball and Chain</string>
        <key>Artist</key>
        <string>Social Distortion</string>
        <key>Album</key>
        <string>Social Distortion</string>
        <key>iTunesSongID</key>
        <integer>197986000</integer>
        <key>iTunesStorefrontID</key>
        <integer>143441</integer>
        <key>StationFrequency</key>
        <string>104.9</string>
        <key>StationCallLetters</key>
        <string>KCNL</string>
        <key>PodcastFeedURL</key>
```

```
            <integer>0</integer>
            <key>StationURL</key>
            <string>http://www.channel1049.com</string>
            <key>Genre</key>
            <string>(17) Rock</string>
            <key>TimeStamp</key>
            <date>2007-04-16T00:48:45Z</date>
            <key>ProgramNumber</key>
            <integer>1</integer>
            <key>iTunesAffiliateID</key>
            <string>00-12uR34oIcpQ</string>
        </dict>
    </array>
</dict>
</plist>
```

A.9.3 Satellite Radio Sample

The following is a sample satellite radio plist-formatted tag file showing a header and one stored tag:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Inc//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
   <key>MajorVersion</key>
   <integer>1</integer>
   <key>MinorVersion</key>
   <integer>1</integer>
   <key>ManufacturerID</key>
   <integer>17</integer>
   <key>ManufacturerName</key>
   <string>Acme</string>
   <key>DeviceName</key>
   <string>AC-HDR100</string>
   <key>MarkedTracks</key>
   <array>
       <dict>
           <key>Name</key>
           <string>Times Like These</string>
           <key>Artist</key>
           <string>Foo Fighters</string>
           <key>iTunesSongID</key>
           <integer>6906304</integer>
           <key>StationFrequency</key>
           <string>9</string>
           <key>StationCallLetters</key>
           <string>90's on 9</string>
           <key>Genre</key>
           <string>(20) Alternative</string>
           <key>TimeStamp</key>
           <date>2007-04-16T00:35:42Z</date>
           <key>iTunesAffiliateID</key>
           <string>00-12uR34oIcpQ</string>
           <key>UnknownData</key>
           <data>MDcwNDEyMzQwMTAyMTEwNDA4MTJhYjU2Z2g=</data>
       </dict>
```

```
   </array>
</dict>
</plist>
```

B. Nike + iPod Cardio Accessory Design

IMPORTANT: This technology is deprecated. Support for it is not guanteed in future Apple devices and it must not be used in new accessory designs.

As described in "Nike + iPod Cardio Equipment System" (page 85), accessories that support Apple's Nike + iPod cardio equipment feature let the user of an Apple device record workout data gathered by an attached sports or exercise device, while at the same time enjoying entertainment from the Apple device.

B.1 Cardio Equipment Requirements

This section specifies the requirements for cardio equipment that works with the Nike + iPod system.

B.1.1 Determining Apple Device Support for Cardio Equipment

To use the Nike + iPod feature, the cardio equipment must first determine that the connected Apple device supports the required minimum lingo versions shown in Table B-1 (page 501).

Table B-1 Lingo version support

Lingo	Minimum Required Protocol Version
0x09: Sports	v1.01
0x0C: Storage	v1.02

If the minimum required protocol versions are supported by the connected Apple device, the cardio equipment must next identify itself as supporting those lingoes; see "Identification Procedure" (page 502).

The final step is for the cardio equipment to query the Sports lingo capabilities of the Apple device with a `GetiPodOptionsForLingo` command and then parse the resulting `RetiPodOptionsForLingo` command for the Nike + iPod Cardio Equipment bit (bit 1); see "Command 0x4C: RetiPodOptionsForLingo" (page 192).

Note: All Sports lingo commands require authentication. The cardio equipment must not send the `GetiPodOptionsForLingo` command until after the authentication process has begun. See "Apple Device Authentication of the Accessory" (page 106) for further information.

B.1.2 Identification Procedure

The cardio equipment must perform an identification procedure by which it discovers the capabilities of the connected Apple device. If it attempts to identify for a lingo not supported by the attached Apple device, the identification fails and the Apple device displays an error message.

The following identification procedure is recommended:

1. Send a `StartIDPS` command. If the Apple device responds with an `iPodAck` command passing 0x04 (Bad Parameter), the accessory must send `IdentifyDeviceLingoes` instead, with `deviceID`=0x0000 and identifying only the General lingo with no options.
2. Query the Apple device for the version numbers of all necessary lingoes.
3. Based on the responses to the query, form appropriate FID values, declare them for the required lingoes, and request authentication if necessary.
4. End the IDPS process by sending `EndIDPS`.

B.1.2.1 Sample Identification Processes

This section presents two sample identification sequences for a cardio equipment accessory. In both cases the sequences begin with the accessory sending a `StartIDPS` command. However, in both cases the Apple device does not support IDPS, so the accessory reverts to sending `IdentifyDeviceLingoes`. For examples of identification sequences in which IDPS is supported, see "Sample Identification Sequences" (page 389).

For a first example, consider a piece of cardio equipment that is intended to offer simple Apple device playback control (play/pause, next/previous track, etc.), display information about the currently playing track, and log workout data to the Apple device. To achieve these goals, the cardio equipment needs access to four iAP lingoes:

- Simple Remote
- Display Remote
- Sports
- Storage

The example in Table B-2 (page 502) shows the command traffic exchanged between the cardio equipment and an iPod Classic running software v1.1.1.

Table B-2 Sample identification process 1

cardio equipment	iPod Classic (v1.1.1)
`StartIDPS`	

cardio equipment	iPod Classic (v1.1.1)
	ACK(StartIDPS, 0x04 = Bad Parameter)
IdentifyDeviceLingoes (Lingoes: 0x0001, Options: 0x00, DeviceID: 0x0000)	
	ACK(IdentifyDeviceLingoes, Success)
RequestTransportMaxPayloadSize	
	ReturnTransportMaxPayloadSize(maxSize)
RequestLingoProtocolVersion(0x02)	
	ReturnLingoProtocolVersion(0x02, v1.02)
RequestLingoProtocolVersion(0x03)	
	ReturnLingoProtocolVersion(0x03, v1.05)
RequestLingoProtocolVersion(0x09)	
	ACK(RequestLingoProtocolVersion, Bad Parameter) (This Apple device does not support the Sports lingo.)
RequestLingoProtocolVersion (0x0C)	
	ReturnLingoProtocolVersion(0x0C, v1.01) (This Apple device does not support the required version of the Storage lingo.)
IdentifyDeviceLingoes(Lingoes:0x000D, Options:0x02, DeviceID:0x0200)	
	ACK(IdentifyDeviceLingoes, Success)
	GetAccessoryAuthenticationInfo()

In this first example, the cardio equipment determines that the attached Apple device does not support the proper versions of the Sports and Storage lingoes and thus does not attempt to identify for those lingoes. Display Remote and Simple Remote are both supported, and the cardio equipment properly identifies and authenticates for those lingoes.

A more successful example is shown in Table B-3 (page 503). It lists the command traffic exchanged between the cardio equipment and a 3G iPod nano updated to support the Nike + iPod feature.

Table B-3 Sample identification process 2

Cardio equipment	3G iPod nano (version 1.1.2)
StartIDPS	

Cardio equipment	3G iPod nano (version 1.1.2)
	ACK(StartIDPS, 0x04 = Bad Parameter)
IdentifyDeviceLingoes (Lingoes:0x0001, Options:0x00, DeviceID:0x0000)	
	ACK(IdentifyDeviceLingoes, Success)
RequestTransportMaxPayloadSize	
	ReturnTransportMaxPayloadSize(maxSize)
RequestLingoProtocolVersion(0x02)	
	ReturnLingoProtocolVersion(0x02, v1.02)
RequestLingoProtocolVersion(0x03)	
	ReturnLingoProtocolVersion(0x03, v1.05)
RequestLingoProtocolVersion(0x09)	
	ReturnLingoProtocolVersion(0x09, v1.01)
RequestLingoProtocolVersion(0x0C)	
	ReturnLingoProtocolVersion(0x0C, v1.02)
IdentifyDeviceLingoes(Lingoes:0x120D, Options:0x02, DeviceID: 0x0200)	
	ACK(IdentifyDeviceLingoes, Success)
	GetAccessoryAuthenticationInfo()

In this second example, the Apple device supports the proper minimum protocol versions for each of the necessary lingoes. The cardio equipment identifies for all of those lingoes and operation proceeds normally.

B.1.3 Declaring Cardio Equipment Support to the Apple Device

In the future, Apple devices may change their user interface elements when attached to Nike + iPod cardio equipment. Hence, the cardio equipment must declare its support of the Nike + iPod feature. This is also necessary so that the Apple device can differentiate between types of Sports lingo accessories (e.g. between a Nike + iPod Sports Kit Receiver and a stair climber).

The cardio equipment declares itself as such by supporting the Sports lingo `GetAccessoryCaps` command and replying to it with a `RetAccessoryCaps` command with `capsMask` bit 9 = 1, indicating to the Apple device that the cardio equipment supports the required commands.

B.1.4 Accessing User Data

Nike + iPod compatible cardio equipment can access certain user data stored on an Apple device. See "Sports Lingo commands" (page 335) for detailed information on the commands available for retrieving and/or setting the available user data fields.

Proper use of this user data requires that the cardio equipment must:

1. Always get/set data for the current `userIndex`. It is assumed that the Apple device is already configured for the correct `userIndex`.
2. Write `userData` fields back to the Apple device if, and only if, the user entered new or updated information via the cardio equipment user interface. For example, if the cardio equipment uses a default value for weight during a workout, that weight must not be written back to the `userData` on the Apple device.
3. Never display the name field for `userIndex` = 0. That value is reserved for the default new or unknown user; consequently, the associated name field is invalid. The `userData` name fields for other `userIndex` values should be valid.

B.1.5 Recording Workout Data

Workout data is stored on the Apple device as a UTF-8 encoded, XML-formatted file. The cardio equipment is responsible for writing the data directly to the Apple device's file system via the Storage lingo. Appropriate XML elements are written to the file in real time as the workout proceeds.

B.1.5.1 Writing Workout Data

After successfully providing valid accessory authentication information to the attached Apple device, the cardio equipment typically uses the following command sequence to write the workout data file to the Apple device. For details of these commands, see "Lingo 0x09: Sports Lingo" (page 335) and "Lingo 0x0C: Storage Lingo" (page 357).

Table B-4 Writing workout data to the Apple device

Cardio equipment	Apple device (with Nike + iPod feature support)
`SportsLingo: GetiPodCaps()`	
	`SportsLingo: RetiPodCaps()`
`StorageLingo: GetiPodCaps()`	
	`StorageLingo: RetiPodCaps()`
`OpeniPodFeatureFile( featureType:0x02, optionsMask:0x000B, fileData:"</gymData>")`	
	`RetiPodFileHandle()`

Cardio equipment	Apple device (with Nike + iPod feature support)
`WriteiPodFileData()` as many times as needed	
	`iPodAck(WriteiPodFileData, status)` for each `WriteiPodFileData` command
`CloseiPodFile()`	
	`iPodAck(CloseiPodFile, ackStatus)`

The cardio equipment starts by sending a `GetiPodCaps` command to retrieve the Sports lingo capabilities of the Apple device. The Apple device responds with a `RetiPodCaps` command.

If the Apple device indicates that it supports the Nike + iPod feature, the cardio equipment next sends a `GetiPodCaps` command to determine the Storage lingo capabilities of the Apple device. The Apple device responds with a `RetiPodCaps` command containing the following data:

- `totalSpace`: the amount of storage on the Apple device, including space currently in use
- `maxFileSize`: the largest possible size of a file on the Apple device
- `maxWriteSize`: the largest amount of data that can be written to the Apple device in a single `WriteiPodFileData` command
- `majorVersion` and `minorVersion`: the version number of the Storage lingo protocol (currently 1.02)

The cardio equipment is required to honor the `maxWriteSize` limitation of the Apple device; it must never send a `WriteiPodFileData` command with a data payload larger than that `maxWriteSize` value.

To create a cardio workout file on the Apple device, the cardio equipment sends an `OpeniPodFeatureFile` command with `feature` = 0x02, `options` = 0x000B, and `fileData` equal to the UTF-8 representation of the string necessary to close the root element: "</gymData>" or 0x3C 0x2F 0x67 0x79 0x6D 0x44 0x61 0x72 0x61 0x3E. On receiving this command, the Apple device performs the following steps:

1. Creates a new workout file in `/iPod_Control/Device/Trainer/Workouts/Empeds/`*USERNAME*`/latest/` (where *USERNAME* is the name of the current `userIndex`)
2. Sends a `RetiPodFileHandle` command with a handle to the new file

Next, the cardio equipment must use a series of `WriteiPodFileData` commands to write the following elements to the file:

the XML version header line
root element opening tag: <gymData>
<vers>
<equipmentInfo>
<userInfo>
<template> (optional)
<interval> with <event> "start" or "continue" and initial values for all child elements

See "XML Element Formats" (page 508) for details about valid XML elements.

As the workout progresses, the cardio equipment is required to write <interval> XML elements on a real-time basis at least every 10 seconds. Certain child elements of <interval> (the <incline> element, for example) are required only if their value has changed since the previous <interval> element .

Note: All <interval> elements written to file must be complete and closed. Because a user can detach the Apple device at any time during the process, <interval> elements must not be broken across multiple `WriteiPodFileData` commands.

At the end of the workout, the cardio equipment must write the <workoutSummary> element to the file and then send the `CloseiPodFile` command. Upon receiving the `CloseiPodFile` command, the Apple device performs the following actions (assuming the proper option bits were set in `OpeniPodFeatureFile`):

1. Append the <ipodInfo> XML element to file.
2. Write the specified `fileData` to close the root element.
3. Close the XML file.
4. Compute and insert an XML <Signature> element in to file.
5. Send an `iPodAck` command acknowledging success.

While performing these steps, the Apple device is unresponsive to further iAP commands. Signature calculation requires many seconds for long workouts. The cardio equipment must wait for the `CloseiPodFile` command to fully complete before sending any new iAP commands. The `CloseiPodFile` command is complete when the Apple device returns an `iPodAck` with any status other than command pending (0x06).

B.1.5.2 Workout Events

Events that occur as part of a normal workout must be tracked and recorded to the XML file as part of the <event> child element of <interval>. There are five event types to be recorded:

- **start:** the first <interval> of a new workout
- **continue:** the first <interval> of a workout already in progress
- **pause:** when a user pauses the active workout
- **resume:** when a user resumes the workout
- **end:** the final <interval> of a completed workout

Workout time must not elapse between a pause event and a resume event; the <sec> children of both the pause and the resume <interval> elements must be identical. Continue events are used to mark a file that was opened after the workout has already started; some amount of <interval> data from the beginning of the workout is not present in the file.

B.1.5.3 File System Full

The Apple device will determine whether or not enough disk space is available to record and sign the workout data. When the Apple device's file system no longer contains enough free space, the Apple device will respond with `iPodAck` = (0x03) (out of resources) to any `OpeniPodFeatureFile` or `WriteiPodFileData` commands. If this occurs during an `OpeniPodFeatureFile` command, the cardio equipment must notify the user that the Apple device is full and that the workout will not be recorded.

If an "out of resources" `iPodAck` is received in response to a `WriteiPodFileData`, the cardio equipment must close the current workout file without writing an end <interval> or <workoutSummary> and must alert the user that the Apple device is full and that the workout is no longer being recorded.

B.1.5.4 File Writing Error Recovery

Cardio equipment accessories must properly handle unexpected failures during the file writing process. In the event of a failure while writing data to the Apple device (as indicated by lack of an `ackStatus=Success` in response to `WriteiPodFileData`), the cardio equipment should first retry the operation. If the operation fails twice, the current workout file must be closed and the cardio equipment must attempt to open a new workout file.

B.1.5.5 Recording a Workout Already in Progress

When opening a new workout data file mid-workout, the standard procedure for creating a new workout file must be followed. The difference between a standard workout file and one started in the middle of a workout is that the first <interval> element will have a nonzero <sec> value and an <event> continue (instead of start) value. It is the responsibility of the end consumer of the data (such as `nikeplus.com`) to recognize a workout file that started mid-workout and act appropriately.

B.1.6 XML Element Formats

All file data must be well-formed and valid XML. Character entities are not currently supported by the Apple device. All special characters (&, <, >) used as element data content must be encapsulated as part of a CDATA section. For human readability, it is recommended that each element close tag be followed by a newline character (0x0A). XML elements are required to appear in the order shown in Table B-5 (page 509) with two exceptions:

- The <userInfo> element may appear at any point in the file.
- The <template> element may appear multiple times and at any point in the file.

All <interval> elements must be written in chronological order (ascending <sec> values). All distances must be recorded to 0.01 km resolution. All kilocalorie values must be recorded to 0.01 kCal resolution. In situations where the user is not moving and the workout speed is essentially zero, the <pace> element should be set to 0 instead of to infinity.

Table B-5 XML element usage

Element	Format	Example	Tread-mill	Ellip-tical	Stepper	Bike	Other
Header							
XML Header		<?xml version="1.0" encoding="UTF-8"?>	Req	Req	Req	Req	Req
Root Element Open		<gymData>	Req	Req	Req	Req	Req
File Format Version	integer	<vers>1</vers>	Req	Req	Req	Req	Req
Equipment Info							
		<equipmentInfo>	Req	Req	Req	Req	Req
Manufacturer ID	integer converted to hexadecimal	<manufacturerID> A5C57 </manufacturerID> (integer = 678999; see Note below)	Req	Req	Req	Req	Req
Manufacturer Name	string	<manufacturerName> Acme </manufacturerName>	Req	Req	Req	Req	Req
Equipment Type	string enum: Treadmill, Elliptical, Stepper, Bike, or Other	<type>Treadmill </type>	Req	Req	Req	Req	Req
Equipment Model	string	<model> ExerPro 5000 </model>	Req	Req	Req	Req	Req
Equipment Serial No.	string	<serialNumber> XYZ012345TG88 </serialNumber>	Opt	Opt	Opt	Opt	Opt
Gym Name	string	<gymName> GloboGym </gymName>	Opt	Opt	Opt	Opt	Opt
Gym Location	string	<gymLocation> Cupertino, CA </gymLocation>	Opt	Opt	Opt	Opt	Opt
		</equipmentInfo>	Req	Req	Req	Req	Req
User Info							

Element	Format	Example	Tread-mill	Ellip-tical	Stepper	Bike	Other
		<userInfo>	Req	Req	Req	Req	Req
User Name	string	<name>Jane Doe </name>	Opt	Opt	Opt	Opt	Opt
Weight	float kilograms	<kg>53.5</kg>	Req	Req	Req	Opt	Opt
Gender	string enum: female or male	<gender>female </gender>	Opt	Opt	Opt	Opt	Opt
		</userInfo>	Req	Req	Req	Opt	Opt
Workout Template							
		<template>	Opt	Opt	Opt	Opt	Opt
Template Name	string	<templateName>Hills </templateName>	Opt	Opt	Opt	Opt	Opt
Caloric Goal	float kilocalories	<kCal>200.00</kCal>	Opt	Opt	Opt	Opt	Opt
Time Goal	integer seconds	<sec>900</sec>	Opt	Opt	Opt	Opt	Opt
Distance Goal	float kilometers	<km>10.00</km>	Opt	Opt	N/A	Opt	Opt
	float floors	<floors>30.0</floors>	N/A	N/A	Opt	N/A	Opt
Speed Goal	integer seconds per kilometer	<pace>450</pace>	Opt	N/A	N/A	N/A	Opt
	integer steps/strides per minute	<spm>40</spm>	N/A	Opt	Opt	N/A	Opt
	integer revolutions per minute	<rpm>60</rpm>	N/A	N/A	N/A	Opt	Opt
Heart Rate Goal	integer beats per minute	<bpm>140</bpm>	Opt	Opt	N/A	Opt	Opt
		</template>	Opt	Opt	Opt	Opt	Opt
Interval Data (written at least every 10 seconds)							
		<interval>	Req	Req	Req	Req	Req
Event Type	string enum: start, continue, pause, resume, or end	<event>pause </event>	Opt	Opt	Opt	Opt	Opt

Element	Format	Example	Tread-mill	Ellip-tical	Stepper	Bike	Other
Current Elapsed Time	integer seconds	<sec>1200</sec>	Req	Req	Req	Req	Req
Current Calories	float kilocalories	<kCal>230.12</kCal>	Req	Req	Req	Req	Req
Current Distance	float kilometers	<km>3.25</km>	Req	Req	N/A	Req	Opt
	float floors	<floors>11.0</floors>	N/A	N/A	Req	N/A	Opt
Current Speed	integer seconds per kilometer	<pace>430</pace>	Req	N/A	N/A	N/A	Opt
	integer steps/strides per minute	<spm>40</spm>	N/A	Req	Req	N/A	Opt
	integer revolutions per minute	<rpm>55</rpm>	N/A	N/A	N/A	Req	Opt
Current Heart Rate	integer beats per minute	<bpm>101</bpm>	Opt	Opt	Opt	Opt	Opt
Incline	float percent	<incline>10.1 </incline>	Required only if value has changed				
Resistance/Effort	integer level	<level>3</level>	Required only if value has changed				
		</interval>	Req	Req	Req	Req	Req
Workout Summary							
		<workoutSummary>	Req	Req	Req	Req	Req
Total Elapsed Time	integer seconds	<sec>2450</sec>	Req	Req	Req	Req	Req
Total Calories	float kilocalories	<kCal>342.34</kCal>	Req	Req	Req	Req	Req
Total Distance	float kilometers	<km>10.15</km>	Req	Req	N/A	Req	Opt
	float floors	<floors>11.0</floors>	N/A	N/A	Req	N/A	Opt
Average Speed	integer seconds per kilometer	<pace>430</pace>	Req	N/A	N/A	N/A	Opt
	integer steps/strides per minute	<spm>40</spm>	N/A	Req	Req	N/A	Opt

Element	Format	Example	Tread-mill	Ellip-tical	Stepper	Bike	Other
	integer revolutions per minute	<rpm>55</rpm>	N/A	N/A	N/A	Req	Opt
Average Heart Rate	integer beats per minute	<bpm>140</bpm>	Opt	Opt	Opt	Opt	Opt
		</workoutSummary>	Req	Req	Req	Req	Req
Apple Device Info							
		<ipodInfo>	Written by Apple device				
File Open Date/Time	date/time format w/ offset	<openTime> 2007-12-17T 17:15:17-08:00 </openTime>	Written by Apple device				
File Close Date/Time	date/time format w/ offset	<closeTime> 2007-12-17T 17:56:07-08:00 </closeTime>	Written by Apple device				
Apple device Model	string	<model>MA980 </model>	Written by Apple device				
Apple Device Software Version	string	<softwareVersion> 1.1.2 </softwareVersion>	Written by Apple device				
Apple device Serial No.	string	<serialNumber> 4H802998TVS </serialNumber>	Written by Apple device				
		</ipodInfo>	Written by Apple device				
XML Signature							
Apple device signature		<Signature> . . . </Signature>	Written by Apple device				
Root Close							
		</gymData>	Written by Apple device				

Note: The Manufacturer ID element is the hexadecimal equivalent of the first part of a product ID assigned by Apple, which identifies the licensee's account. For example, if Apple assigns a product ID of 678999-12345, the Manufacturer ID element must be contain the hexadecimal equivalent of 678999, which is A5C57. To obtain a product ID number, the licensee must submit a product plan to Apple's MFi portal.

B.2 User Interface Requirements

This section sets forth the requirements for the cardio equipment's user interface. An example of a typical user interface process, from the perspective of the cardio equipment, is shown in Figure B-1 (page 515).

B.2.1 The Apple Device Does Not Support Nike + iPod

The cardio equipment must determine whether the attached Apple device supports the Nike + iPod feature; see "Determining Apple Device Support for Cardio Equipment" (page 501). If the attached Apple device does not support the Nike + iPod feature, then the cardio equipment must display this information to the user. See Table B-6 (page 514) for the approved UI string.

B.2.2 Deciding to Record a Workout to the Apple Device

The cardio equipment must first check the Apple device for the user's Workout Recording Preference, using the Sports lingo. If the user's recording preference is set to Never then the workout must not be recorded. If the user's preference is set to Always, Ask, or Not Set, then the cardio equipment must present an option to the user. There are two approved forms for this option:

- Recommended: allow user to opt out. Default to recording the workout, but present an option for the user to cancel the recording.
- Allowed: present a dialog prompting the user to choose whether or not they would like the workout to be recorded.

B.2.3 The Apple Device is Full

In the event that the cardio equipment is unable to record a workout to the Apple device because of insufficient disk space, the cardio equipment must present a message to the user alerting them to this problem. See Table B-6 (page 514) for the approved UI string. Note that this problem may occur either when trying to open a workout file or mid-workout, if the Apple device's file system becomes full.

B.2.4 User Weight

Workouts are displayed at `nikeplus.com` using a metric known as Cardio Miles. Cardio Miles are a conversion from workout calories to equivalent running miles. The key metric recorded in most cardio equipment workouts is the calories burned. To accurately calculate calories burned, the user must be required to enter a weight value.

Because a user's weight may change over time, the equipment must always present the user with an opportunity to enter or adjust the weight value. The enter-weight UI on the cardio equipment must start with the weight value retrieved from Apple device via the `GetUserData` command. It is acceptable for the cardio equipment UI to automatically accept the displayed weight if the user fails to enter or adjust the weight after a length of time has passed.

If the user enters a weight different from that retrieved from the Apple device, the cardio equipment must write the new weight value back to Apple device via the `SetUserData` command.

B.2.5 Recording Indicator

Cardio equipment must always present a visible indicator to the user that the workout is being successfully recorded to Apple device. This indicator must be present as soon as the workout recording begins and then disappear when the workout recording ends (either because the workout is finished or the Apple device has been detached from the equipment).

Whenever the user pauses a workout, the recording indicator must flash on and off until the workout either resumes normally or is terminated.

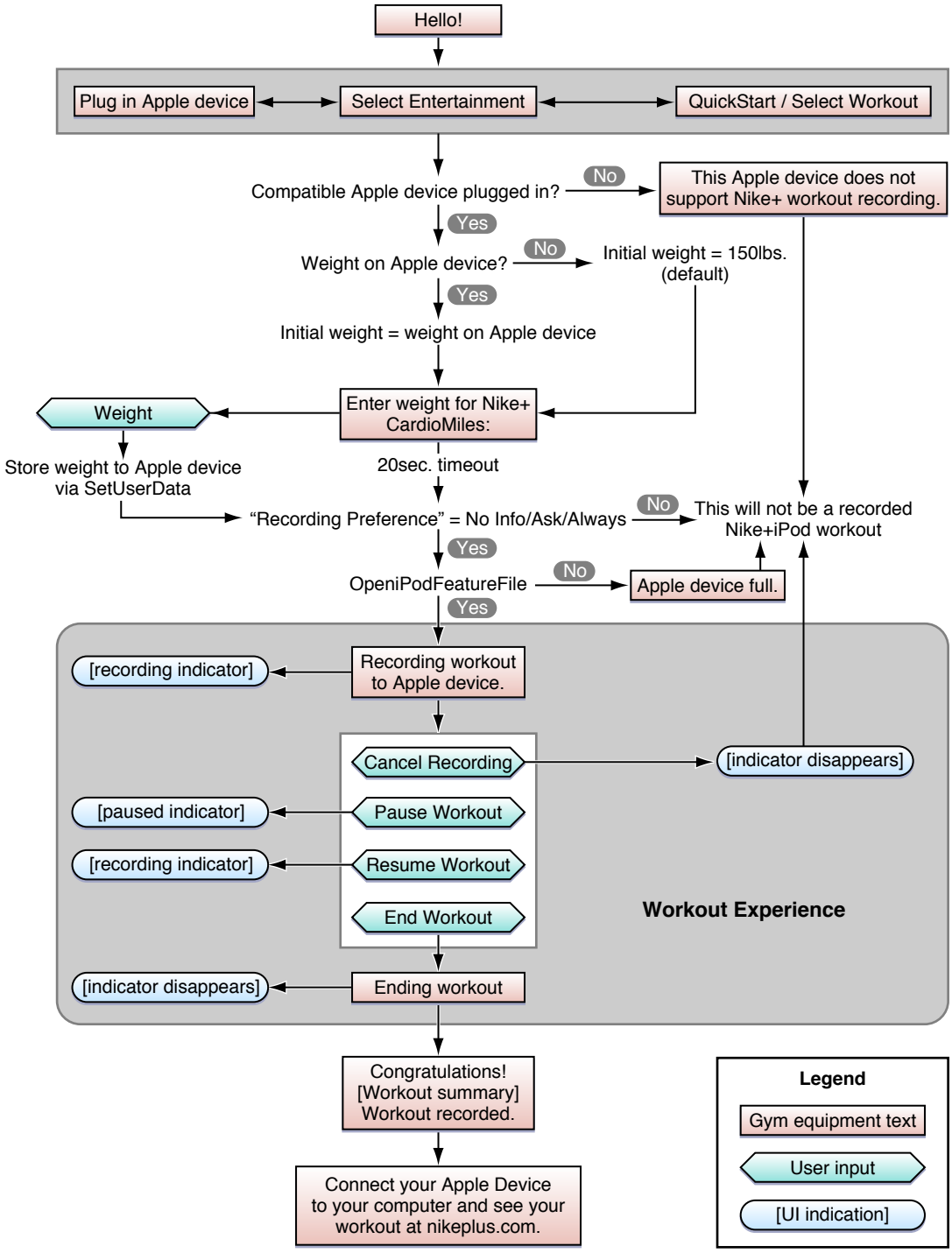
B.2.6 Workout Completed

At the end of a workout, the cardio equipment must present two messages to the user. The first is a congratulatory message stating that the workout has been successfully recorded and showing a summary of the workout data. The second message gives brief instruction on how to view the user's workout statistics at the `nikeplus.com` website. Table B-6 (page 514) lists all the approved UI strings for display by Nike + iPod cardio equipment.

Table B-6 Approved user interface messages

Event	Approved UI string	Short UI string
Apple Device Does Not Support the Nike + iPod System	This Apple device does not support Nike + iPod workout recording.	This Apple device does not support workout recording.
The Apple device is full	This Apple device is full.	Apple device full.
Workout summary	Congratulations! <Workout summary> Workout recorded.	Congratulations! <Workout summary> Workout recorded.
Upload instructions	Connect your Apple device to your computer to see your workout at nikeplus.com.	See your workout at nikeplus.com.

Figure B-1 Sample user experience flow chart



B.3 Sample Command Sequence

Table B-7 (page 516) shows a sample sequence of IDPS commands that implements the Nike + iPod cardio equipment system in an accessory.

Table B-7 Cardio equipment command sequence using IDPS

Step	Accessory command	Apple device command	Comment
1	`StartIDPS`		no params
2		`iPodAck`	acknowledging 'Success (OK)' to command 'General Lingo::StartIDPS'
If the `iPodAck` reply to `StartIDPS` returns a status of 0x00, the accessory proceeds to Step 3. Otherwise, ■ If the accessory receives no `iPodAck` command within 500 ms, it may retry Step 1 at 500 ms intervals as long as Apple Device Detect (pin 30 of the 30-pin connector) is low and Accessory Power (pin 13) is high. Accessories that don't monitor these signals (such as Bluetooth accessories or USB accessories without a 191 kΩ ID resistor) may retry without checking them. ■ If the accessory receives an `iPodAck` status of 0x04 (Bad Parameter), the Apple device does not support IDPS. Within 800 ms the accessory must send an `IdentifyDeviceLingoes` command with its lingo mask set for only the General lingo, with no options, and with a Device ID of 0x0. See "Canceling a Current Authentication Process With IdentifyDeviceLingoes" (page 135). A sample command sequence is listed in Table C-55 (page 573). ■ If the accessory receives any other nonzero `iPodAck` status, it must not send any more iAP commands until it has been disconnected. Upon reconnection it must restart identification at Step 1.			
3	`RequestTransport-MaxPayloadSize`		requesting transport maximum payload size
4		`ReturnTransport-MaxPayloadSize`	returning transport maximum payload size
5	`GetiPodOptions-ForLingo`		getting options for General Lingo
6		`RetiPodOptions-ForLingo`	returning options of 000000000003F3FF (Line out usage | Video output | NTSC video signal format | PAL video signal format | Composite video out connection | S-Video video out connection | Component video out connection | Closed captioning (video) | Video aspect ratio 4:3 (fullscreen) | Video aspect ratio 16:9 (widescreen) | reserved | app communication capable | Apple device notifications) for General Lingo
7	`GetiPodOptions-ForLingo`		getting options for Sports Lingo

Step	Accessory command	Apple device command	Comment
8		`RetiPodOptions-ForLingo`	returning options of 0000000000000002 (Nike + iPod cardio equipment) for Sports Lingo
0	`GetiPodOptions-ForLingo`		getting options for Storage Lingo
10		`RetiPodOptions-ForLingo`	returning options of 0000000000000003 (iTunes tagging | Nike + iPod cardio equipment) for Storage Lingo
11	`SetFIDTokenValues`		setting 12 FID tokens ; IdentifyToken = (lingoes: 0/9/12 | options 0x00000002 | device ID 0x00000200); AccCapsToken = 0x0000000000000805; AccInfoToken = Acc name (FitBicycle 2010); AccInfoToken = Acc FW version (v1.0.1); AccInfoToken = Acc HW version (v1.0.0); AccInfoToken = Acc manufacturer (Apple); AccInfoToken = Acc model number (MA1234LL/A); iPodPreferenceToken = (setting 'used' for preference class 'line out usage' with restore on exit 'selected'); iPodPreferenceToken = (setting 'on' for preference class 'video out setting' with restore on exit 'selected'); iPodPreferenceToken = (setting 'fullscreen' for preference class 'screen configuration' with restore on exit 'selected'); iPodPreferenceToken = (setting 'NTSC' for preference class 'video format setting' with restore on exit 'selected'); iPodPreferenceToken = (setting 'component' for preference class 'video connection' with restore on exit 'selected')

Step	Accessory command	Apple device command	Comment
12		AckFIDTokenValues	12 ACKs for FID tokens ; IdentifyToken = accepted; AccCapsToken = accepted; AccInfoToken = (Acc name) accepted; AccInfoToken = (Acc FW version) accepted; AccInfoToken = (Acc HW version) accepted; AccInfoToken = (Acc manufacturer) accepted; AccInfoToken = (Acc model number) accepted; iPodPreferenceToken = (line out usage) accepted; iPodPreferenceToken = (video out setting) accepted; iPodPreferenceToken = (screen configuration) accepted; iPodPreferenceToken = (video format setting) accepted; iPodPreferenceToken = (video connection) accepted
13	EndIDPS		status 'finished with IDPS; proceed to authentication'
14		IDPSStatus	status 'ready for auth'
15		GetAccessory-AuthenticationInfo	no params
16	RetAccessory-AuthenticationInfo		returning auth protocol v2.0; current section index: 0; maximum section index: 1; cert data ...
17		iPodAck	acknowledging 'Success (OK)' to command 'General Lingo::RetAccessoryAuthenticationInfo'
18	RetAccessory-AuthenticationInfo		returning auth protocol v2.0; current section index: 1; maximum section index: 1; cert data ...
19		AckAccessory-AuthenticationInfo	acknowledging 'auth info supported'
20		GetAccessoryCaps	no params
21		GetAccessory-Authentication-Signature	offering challenge '...' with retry counter 1
22	RetAccessory-Authentication-Signature		returning signature '...'
23		AckAccessory-AuthenticationStatus	acknowledging authentication status 'Success (OK)'

Step	Accessory command	Apple device command	Comment
24		GetAccessoryCaps	requesting accessory Sports lingo capabilities; no params
25		GetAccessoryCaps	requesting accessory Storage lingo capabilities; no params
26	RetAccessoryCaps		returning 'Gym equipment command support'
27	RetAccessoryCaps		returning 'file system type 'none'; lingo v1.02'
28	GetiPodCaps		requesting Apple device Sports lingo capabilities; no params
29		RetiPodCaps	returning 'Gym equipment support | User data support' with userCount of 1
30	GetiPodCaps		requesting Apple device Storage lingo capabilities; no params
31		RetiPodCaps	returning 'total space 2147483648; max file size 1073741824; max write size 500; v1.02'
32	GetUserData		requesting 'Gender'
33		RetUserData	returning 'No information' for data type 'Gender'
34	GetUserData		requesting 'Age'
35		RetUserData	returning '26 years' for data type 'Age'
36	GetUserData		requesting 'Name'
37		RetUserData	returning 'NewUser' for data type 'Name'
38	GetUserData		requesting 'Weight'
39		RetUserData	returning '92.0 kg' for data type 'Weight'
40	OpeniPodFeatureFile		opening feature type 'Gym Equipment Workout' with options mask 'file data | ipodInfo | XML signature' and file data:</gymData>;
41		RetiPodFileHandle	returning file handle in response to command 'Storage Lingo::OpeniPodFeatureFile'

Step	Accessory command	Apple device command	Comment
42	WriteiPodFileData		writing 39 bytes at offset 0 and returned file handle: <?xml version=""1.0"" encoding=""UTF-8""?>
43		iPodAck	acknowledging 'Success' to command 'Storage Lingo::WriteiPodFileData' with file handle 0
44	WriteiPodFileData		writing 10 bytes at offset 39 and handle 0: <gymData>;
45		iPodAck	acknowledging 'Success' to command 'Storage Lingo::WriteiPodFileData' with file handle 0
46	WriteiPodFileData		writing 15 bytes at offset 49 and handle 0: <vers>1</vers>;
47		iPodAck	acknowledging 'Success' to command 'Storage Lingo::WriteiPodFileData' with file handle 0
48	WriteiPodFileData		writing 166 bytes at offset 64 and handle 0: <equipmentInfo> < manufacturerID>123A3A27 </manufacturerID> < manufacturerName>Scotty </manufacturerName> < type>Bike</type> < model>Bike for Scotty</model> </equipmentInfo>
49		iPodAck	acknowledging 'Success' to command 'Storage Lingo::WriteiPodFileData' with file handle 0
50	WriteiPodFileData		writing 36 bytes at offset 230 and handle 0: <userInfo> < kg>90</kg> </userInfo>
51		iPodAck	acknowledging 'Success' to command 'Storage Lingo::WriteiPodFileData' with file handle 0
52	WriteiPodFileData		writing 74 bytes at offset 266 and handle 0: <template> < templateName>Goal </templateName> < kCal>500</kCal> </template>
53		iPodAck	acknowledging 'Success' to command 'Storage Lingo::WriteiPodFileData' with file handle 0

Step	Accessory command	Apple device command	Comment
54	WriteiPodFileData		writing 121 bytes at offset 340 and handle 0: <interval> < event>start</event> < sec>0</sec> < kCal>0</kCal> < km>0</km> < rpm>0</rpm> < bpm>0</bpm> < level>1</level> </interval>
55		iPodAck	acknowledging 'Success' to command 'Storage Lingo::WriteiPodFileData' with file handle 0
56	WriteiPodFileData		writing 88 bytes at offset 461 and handle 0: <interval> < sec>10</sec> < kCal>0</kCal> < km>2</km> < rpm>74</rpm> < bpm>96</bpm> </interval>
57		iPodAck	acknowledging 'Success' to command 'Storage Lingo::WriteiPodFileData' with file handle 0
58	WriteiPodFileData		writing 90 bytes at offset 549 and handle 0: <interval> < sec>20</sec> < kCal>2</kCal> < km>10</km> < rpm>98</rpm> < bpm>163</bpm> </interval>
59		iPodAck	acknowledging 'Success' to command 'Storage Lingo::WriteiPodFileData' with file handle 0
60	WriteiPodFileData		writing 126 bytes at offset 639 and handle 0: <interval> < event>end</event> < sec>30</sec> < kCal>500</kCal> < km>17</km> < rpm>87</rpm> < bpm>172</bpm> < level>2</level> </interval>
61		iPodAck	acknowledging 'Success' to command 'Storage Lingo::WriteiPodFileData' with file handle 0
62	WriteiPodFileData		writing 104 bytes at offset 765 and handle 0: <workoutSummary> < sec>30</sec> < kCal>500</kCal> < km>17</km> < rpm>89</rpm> < bpm>167</bpm> </workoutSummary>
63		iPodAck	acknowledging 'Success' to command 'Storage Lingo::WriteiPodFileData' with file handle 0

Step	Accessory command	Apple device command	Comment
64	`CloseiPodFile`		closing file with handle 0
65		`iPodAck`	acknowledging 'Success' to command 'Storage Lingo::CloseiPodFile' with file handle 0

C. Historical Information

This appendix memorializes the specifications of past Apple device models and iAP protocols. It is included in this specification to provide guidance for developers who need to design accessories compatible with these past technologies.

C.1 Past Protocol Features

Table C-1 (page 523) and Table C-2 (page 525) list the past protocol versions for each lingo and the firmware releases in which they were available.

In the following tables, "NL" indicates that the given lingo was not implemented in the specified model loaded with the specified firmware. "NV" indicates that although the lingo was implemented, its protocol version could not be read from the Apple device. Footnotes are listed below each table.

Table C-1 Past Apple device models, firmware, and lingo versions, table 1

Model	Firmware		Lingoes										
	Version	Date	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x0A	0x0C
3G [1]	2.0.0	04/03	NV	NL	NV	NL	NL	NL	NL	NL	NL	NL	NL
	2.1.0	10/03	NV	NV	NV	NL	1.00	NL	NL	NL	NL	NL	NL
	2.2.0	02/04	NV	NV	NV	NL	1.02	NL	NL	NL	NL	NL	NL
	2.3.0	02/05	NV	NV	NV	NL	1.02	NL	NL	NL	NL	NL	NL
mini	1.0.0	02/04	NV	NL	NV	NL	1.01	NL	NL	NL	NL	NL	NL
	1.1.0	03/04	NV	NL	NV	NL	1.03	NL	NL	NL	NL	NL	NL
	1.2.0	11/04	1.00	NL	1.00	NL	1.05	1.00	NL	NL	NL	NL	NL
	1.3.0	02/05	1.00	NL	1.00	NL	1.05	1.00	NL	NL	NL	NL	NL
	1.4.0	06/05	1.02	NL	1.00	1.01	1.05	1.00	NL	NL	NL	NL	NL
4G	3.0.0	07/04	NV	NV	NV	NL	1.04	NV	NL	NL	NL	NL	NL
	3.0.1	08/04	NV	NV	NV	NL	1.04	NV	NL	NL	NL	NL	NL
	3.0.2	11/04	1.00	1.00	1.00	NL	1.05	1.00	NL	NL	NL	NL	NL
	3.1.0	06/05	1.02	1.00	1.00	1.01	1.05	1.00	NL	NL	NL	NL	NL

Model	Firmware		Lingoes										
	Version	Date	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x0A	0x0C
4G (color display)	1.0.0	10/04	1.00	1.00	1.00	NL	1.05	1.00	NL	NL	NL	NL	NL
	1.1.0	03/05	1.01	1.00	1.00	1.00	1.06	1.00	NL	NL	NL	NL	NL
	1.2.0	06/05	1.02	1.00	1.00	1.01	1.06	1.00	NL	NL	NL	NL	NL
nano	1.0.0	09/05	1.02	NL	1.00	1.02	1.07	1.00	NL	1.00	1.00	NL	NL
	1.1.0	01/06	1.03	NL	1.00	1.03	1.09	1.00	NL	1.00	1.00	NL	NL
	1.1.1	03/06	1.03	NL	1.00	1.03	1.09	1.00	NL	1.00	1.00	NL	NL
	1.2.0	06/06	1.04	NL	1.02	1.04	1.10	1.01	NL	1.00	1.00	1.00 [2]	NL
	1.3	10/06	1.05	NL	1.02	1.05	1.11	1.01	NL	1.00	1.00	1.01	NL
5G	1.0.0	10/05	1.03	1.01	1.01	1.03	1.08	1.00	1.00	1.00	1.00	NL	NL
	1.1.0	01/06	1.03	1.01	1.01	1.03	1.09	1.00	1.00	1.00	1.00	NL	NL
	1.1.1	03/06	1.03	1.01	1.01	1.03	1.09	1.00	1.00	1.00	1.00	NL	NL
	1.1.2	06/06	1.03	1.01	1.01	1.03	1.09	1.00	1.00	1.00	1.00	NL	NL
	1.2.0	09/06	1.05	1.01	1.02	1.05	1.11	1.01	1.00	1.00	1.00	1.01	NL
	1.2.1	11/06	1.05	1.01	1.02	1.05	1.11	1.01	1.00	1.00	1.00	1.01	NL
	1.2.3	11/07	1.06	1.01	1.02	1.05	1.12	1.01	1.00	1.00	1.00	1.01	1.01
2G nano	1.0.0	09/06	1.04	1.01	1.02	1.04	1.10	1.01	NL	1.00	1.00	1.00 [2]	NL
	1.0.1	09/06	1.04	1.01	1.02	1.04	1.10	1.01	NL	1.00	1.00	1.00 [2]	NL
	1.0.2	09/06	1.04	1.01	1.02	1.04	1.10	1.01	NL	1.00	1.00	1.00 [2]	NL
	1.1	10/06	1.04	1.01	1.02	1.04	1.10	1.01	NL	1.00	1.00	1.00 [2]	NL
	1.1.1	10/06	1.04	1.01	1.02	1.04	1.10	1.01	NL	1.00	1.00	1.00 [2]	NL
	1.1.2	02/07	1.04	1.01	1.02	1.04	1.10	1.01	NL	1.00	1.00	1.02	NL
classic	1.0	09/07	1.07	1.01	1.02	1.05	1.12	1.01	NL	1.00	1.00	1.03	1.01
	1.0.1	09/07	1.07	1.01	1.02	1.05	1.12	1.01	NL	1.00	1.00	1.03	1.01
	1.0.2	10/07	1.07	1.01	1.02	1.05	1.12	1.01	NL	1.00	1.00	1.03	1.01
	1.0.3	11/07	1.07	1.01	1.02	1.05	1.13	1.01	NL	1.00	1.00	1.03	1.01
	1.1.1	02/08	1.07	1.01	1.02	1.05	1.13	1.01	NL	1.00	1.00	1.03	1.01

Model	Firmware		Lingoes										
	Version	Date	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x0A	0x0C
	1.1.2	05/08	1.07	1.01	1.02	1.05	1.13	1.01	NL	1.00	1.00	1.03	1.02
3G nano	1.0	09/07	1.07	1.01	1.02	1.05	1.12	1.01	NL	1.00	1.00	1.03	1.01
	1.0.1	09/07	1.07	1.01	1.02	1.05	1.12	1.01	NL	1.00	1.00	1.03	1.01
	1.0.2	10/07	1.07	1.01	1.02	1.05	1.12	1.01	NL	1.00	1.00	1.03	1.01
	1.0.3	11/07	1.07	1.01	1.02	1.05	1.13	1.01	NL	1.00	1.00	1.03	1.01
	1.1	01/08	1.07	1.01	1.02	1.05	1.13	1.01	NL	1.00	1.00	1.03	1.01
	1.1.2	05/08	1.07	1.01	1.02	1.05	1.13	1.01	NL	1.00	1.00	1.03	1.02

[1] Supporting the 3G iPod requires special design considerations. See "The 3G iPod" (page 527) and "Interfacing With the 3G iPod" in *MFi Accessory Hardware Specification*.

[2] Because version 1.00 of Lingo 0x0A contained a bug that was corrected in version 1.01, accessories should attempt Digital Audio only with Apple devices whose Lingo 0x0A version is higher than or equal to 1.01. See Table 4-233 (page 346).

Table C-2 Past Apple device models, firmware, and lingo versions, table 2

Model	Firmware		Lingoes											
	Vers.	Date	0x00	0x01	0x02	0x03	0x04	0x05	0x07	0x08	0x09	0x0A	0x0C	0x0E
iPhone	1.1	09/07	1.07	NL	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	NL	NL
	1.1.1	09/07	1.07	NL	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	NL	NL
	1.1.2	11/07	1.07	NL	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	NL	NL
	1.1.3	01/08	1.07	NL	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	NL	NL
	1.1.4	02/08	1.07	NL	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	NL	NL
	1.1.5	07/08	1.07	NL	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	NL	NL
	2.0	07/08	1.07	NL	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	NL	NL
	2.0.1	08/08	1.07	NL	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	NL	NL
	2.1.0	09/08	1.07	NL	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	1.02	NL
	2.2.1	01/09	1.08	1.02	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	1.02	NL
	3.0.0	06/09	1.09	1.02	1.02	1.05	1.12	1.01	NL	1.00	1.01	1.02	1.02	1.00
	3.1.2	10/09	1.09	1.02	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	1.02	1.00

Model	Firmware		Lingoes											
	Vers.	Date	0x00	0x01	0x02	0x03	0x04	0x05	0x07	0x08	0x09	0x0A	0x0C	0x0E
1G touch	1.1	09/07	1.07	NL	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	NL	NL
	1.1.1	09/07	1.07	NL	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	NL	NL
	1.1.2	11/07	1.07	NL	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	NL	NL
	1.1.3	01/08	1.07	NL	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	NL	NL
	1.1.4	02/08	1.07	NL	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	NL	NL
	1.1.5	07/08	1.07	NL	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	NL	NL
	2.0	07/08	1.07	NL	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	NL	NL
	2.0.1	08/08	1.07	NL	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	NL	NL
	2.1.0	09/08	1.08	NL	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	1.02	NL
	3.0.0	06/09	1.09	1.02	1.02	1.05	1.12	1.01	NL	1.00	1.01	1.02	1.02	1.00
	3.1.2	10/09	1.09	1.02	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	1.02	1.00
iPhone 3G	2.1.0	09/08	1.08	1.02	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	1.02	NL
	2.2.1	01/09	1.08	1.02	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	1.02	NL
	3.0.0	06/09	1.09	1.02	1.02	1.05	1.12	1.01	NL	1.00	1.01	1.02	1.02	1.00
	3.1.2	10/09	1.09	1.02	1.02	1.05	1.12	1.01	NL	1.00	NL	1.02	1.02	1.00
4G nano	1.0.2	09/08	1.08	1.02	1.03	1.05	1.14	1.01	1.01	1.00	1.01	1.03	1.02	NL
	1.0.3	01/09	1.08	1.02	1.04	1.05	1.14	1.01	1.01	1.00	1.01	1.03	1.02	NL
120 GB classic	2.0.0	09/08	1.08	1.02	1.02	1.05	1.13	1.01	1.00	1.00	NL	1.03	1.02	NL
	2.0.1	01/09	1.08	1.02	1.03	1.05	1.13	1.01	1.00	1.00	NL	1.03	1.02	NL
2G touch	2.1.1	09/08	1.08	1.02	1.02	1.05	1.12	1.01	NL	1.00	1.01	1.02	1.02	NL
	2.2.1	01/09	1.08	1.02	1.02	1.05	1.12	1.01	NL	1.01	NL	1.02	1.02	NL
	3.0.0	06/09	1.09	1.02	1.02	1.05	1.12	1.01	NL	1.00	1.01	1.02	1.02	1.00
	3.0.0	06/09	1.09	1.02	1.02	1.05	1.12	1.01	NL	1.00	1.01	1.02	1.02	1.00
iPhone 3GS	3.0.0	06/09	1.09	1.02	1.02	1.05	1.12	1.01	NL	1.00	1.01	1.02	1.02	1.00
	3.1.2	10/09	1.09	1.02	1.02	1.05	1.12	1.01	NL	1.00	1.01	1.02	1.02	1.00
160 GB classic	2.0.3	09/09	1.08	1.02	1.03	1.05	1.13	1.01	1,00	1.00	NL	1.03	1.02	NL

Model	Firmware		Lingoes											
	Vers.	Date	0x00	0x01	0x02	0x03	0x04	0x05	0x07	0x08	0x09	0x0A	0x0C	0x0E
5G nano	1.0.1	09/09	1.08	1.02	1.04	1.05	1.14	1.01	1.01	1.00	1.01	1.03	1.02	NL

C.2 The 3G iPod

The *iPod Accessory Protocol Interface Specification* no longer covers the 3G iPod. The 3G iPod is the white iPod with 4 buttons above a white click wheel, shipped in April, 2003, shown in Table C-12 (page 534). This section is included for historical purposes only.

Table C-3 3G iPod product identification

3G iPod		
	*Product name:* iPod (3rd generation)	*Shipped:* 04/2003
	Compatibility icons:  iPod 3rd generation 10GB 15GB 20GB  iPod 3rd generation 30GB 40GB	
	Connectivity: Dock connector, headphone jack	

C.2.1 General Compatibility With Accessories

3G iPod firmware version 2.0.0 supports only the small packet format with a maximum payload of 127 bytes; it does not support commands that require a larger payload. See "Command Packets" (page 109).

3G iPod firmware versions 2.1.0 and later include limited support for the Accessory Power lingo (Lingo 0x05), but only through the 9-pin audio/remote connector. They do not support this lingo through the 30-pin connector.

C.2.2 Accessory Identification and iAP Commands

The 3G iPod does not support the IDPS process or the `IdentifyDeviceLingoes` command; it supports only the older `RequestIdentify` and `Identify` commands. See "General Lingo Command 0x01: Identify" (page 530). In addition, the 3G iPod does not support authentication.

The 3G iPod does not return a response to an `Identify` command; an attached accessory must use another mechanism to determine whether the iPod has successfully processed the command.

The 3G iPod does not support the Display Remote lingo (Lingo 0x03).

The 3G iPod has limited functionality when executing certain database navigation commands in Extended Interface mode:

- `SelectDBRecord` does not support the track category.
- `RetrieveCategorizedDatabaseRecords` does not support a record count of –1.
- `ResetDBSelection` does not clear the database sort order.

C.2.3 User Interface Restrictions

The 3G iPod does not handle Fast Forward and Rewind notifications properly (see "Command 0x09: RemoteEventNotification" (page 256)).

The 3G iPod does not support podcasts or audiobooks.

C.3 9-Pin Audio/Remote Connector Commands

The commands documented in this section apply only to the 9-pin Audio/Remote connector. For the top connector microphone only, the Apple device sends a `BeginRecord` command when recording is about to begin. The accessory microphone bias, if applicable, should already be present. The Apple device sends an `EndRecord` command when recording is completed. The accessory may remove microphone bias, if applicable, after the `EndRecord` command is received.

The Apple device sends a `BeginPlayback` command when playback is about to begin. The accessory may then turn on its speaker amplifier, if present. Upon receipt of an `EndPlayback` command, the accessory must turn off its speaker amplifier. For all Microphone lingo commands sent by the Apple device, no accessory response is expected.

C.3.1 Command 0x00: BeginRecord

Lingo: 0x01 — Origin: Apple device

The Apple device sends this command to notify the accessory that audio recording has started. The accessory does not return a packet to the Apple device in response to this command. See "Command 0x06: iPodModeChange" (page 536) for more details.

Table C-4 `BeginRecord` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

C.3.2 Command 0x01: EndRecord

Lingo: 0x01 — Origin: Apple device

The Apple device sends this command to notify the accessory that audio recording has ended. The accessory does not return a packet to the Apple device in response to this command. See "Command 0x06: iPodModeChange" (page 536) for more details.

Table C-5 `EndRecord` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

C.3.3 Command 0x02: BeginPlayback

Lingo: 0x01 — Origin: Apple device

The Apple device sends this command to notify the accessory that audio playback has started. The accessory does not return a packet to the Apple device in response to this command. See "Command 0x06: iPodModeChange" (page 536) for more details.

Table C-6 `BeginPlayback` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

C.3.4 Command 0x03: EndPlayback

Lingo: 0x01 — Origin: Apple device

The Apple device sends this command to notify the accessory that audio playback has ended. The accessory does not return a packet to the Apple device in response to this command. See "Command 0x06: iPodModeChange" (page 536) for more details.

Table C-7 `EndPlayback` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		

Value	Bytes	Parameter
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

C.4 Deprecated General Lingo Commands

Note: These commands are deprecated; do not use them in new products.

C.4.1 General Lingo Command 0x01: Identify

Lingo: 0x00 — Origin: Accessory

This command is deprecated. It should be used only by accessories that need to support the 3G iPod. Supporting the 3G iPod requires other special design considerations; see "Interfacing With the 3G iPod" in *MFi Accessory Hardware Specification*.

Accessories that use this command send it to notify the Apple device that an accessory has been attached and to register the lingo it supports. Accessories should identify at boot time and any time they receive "Command 0x00: RequestIdentify" (page 124) from the Apple device. The Apple device does not send an `iPodAck` command in response.

Note: The `Identify` command should be used only over UART transport.

Accessories should follow the identification guidelines in "UART" (page 88) to guarantee they have established communication with the Apple device when using this command. Accessory accessories that support more than one lingo (not including the General lingo) or that plan to use the USB transport should use the IDPS process.

The `Identify` command disables all but free lingoes on the current port. For serial ports, this means lingoes 0x00, 0x02, 0x03, and 0x05 may be used, excluding authenticated commands; the USB port will be able to use only the general lingo, 0x00 (see Table 2-7 (page 104)). Devices that register with this command can use only the General lingo (0x00) commands plus those of the specific lingo that they identified, with a few exceptions. All accessories may use the Simple Remote lingo (0x02) `ContextButtonStatus` command (0x00). If the identified lingo is the Extended Interface lingo (0x04), the accessory may also use the Display Remote lingo (0x03) commands if that lingo is not already being used by another accessory.

If the lingo identified by the `Identify` command can be used by only one accessory at a time, and that lingo is already in use by a different accessory, the command will fail but no command failure `iPodAck` command will be sent to the accessory. The accessory can verify that the `Identify` command has succeeded by sending a free command with valid parameters and checking the Apple device's response. The Apple device will respond with an `iPodAck` response with failure status if the lingo is already in use.

This command performs lingo conflict checking to ensure that single-instance lingoes, such as Display Remote, are used on only one port at a time.

Table C-8 `Identify` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x0N	1	Supported lingo. For example, if the accessory supports the Simple Remote lingo, this byte should be 0x02.
Packet footer specified in "Command Packets" (page 109).		

The `Identify` command has facilities for accessories to draw more than 5 mA power from the Apple device. The `Identify` command sent by such an accessory contains the supported lingo and the optional power bitfields described in Table C-21 (page 538). These bits define the power requirements of the accessory. Table C-20 (page 538) shows the format of an `Identify` command for a high-power accessory.

Table C-9 `Identify` packet for high-power accessories

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x05	1	Supported lingo: Accessory Power lingo
0x00	1	Reserved: set to 0x00
0x02	1	Number of valid bits in the power option flag
0x0N	1	Option flag bits. See Table C-21 (page 538).
Packet footer specified in "Command Packets" (page 109).		

The option flag byte consists of bitfields. At this time, the most significant 6 bits of the option flags are reserved and should be zero. Bit 1 is defined for Apple use only and must be zero. Bit 0 should be 1 if the accessory requires more than 5 mA at any time. This may be the case if it is powered by the Apple device.

Table C-10 Power option bits

Bit	Description
0	Power consumption requirements. Possible values are: 0 = the accessory requires 5 mA or less power from Apple device at all times 1 = the accessory requires more than 5 mA power from Apple device (up to 100mA maximum) during playback operation
1	Reserved. This bit must be set to 0 by external accessories.
2–7	Reserved. Set to 0.

C.4.2 Command 0x05: EnterExtendedInterfaceMode

Lingo: 0x00 — Origin: Accessory

This command is deprecated. Use `SetUIMode` instead.

The accessory sends this command to the Apple device to force it to enter the Extended Interface mode. If the Apple device is already in the Extended Interface mode, it immediately returns a General lingo `iPodAck` command packet, notifying the user that the command was successful. This command may be used only if the accessory requests Lingo 0x04 during its identification process.

Note: This command fails if the Apple device does not detect a valid R_{ID} resistor. See "Accessory Detect and Identify" in *MFi Accessory Hardware Specification*.

If the Apple device needs to switch modes, it returns a General lingo iPodAck Pending command packet, informing the accessory how long it will take the Apple device to switch modes. This is followed by an `iPodAck` packet notifying the accessory that the Apple device successfully changed modes. Accessories should honor the timeout returned by the iPodAck Pending before assuming the original command has failed.

If audio is playing when the Apple device enters Extended Interface mode, the Apple device will pause playback. If video is playing, the Apple device will stop playback.

Table C-11 `EnterExtendedInterfaceMode` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

C.5 Deprecated Lingo 0x01: Microphone Lingo

Note: This lingo is deprecated; do not use it in new products.

The Microphone lingo enables combination microphone and speaker accessory devices to record and play back audio. Initial microphone devices supported one input mode (mono) and one sample rate (8 kHz). The increased Apple device mass storage disk capacities enable the option of supporting a stereo input mode and higher audio sample rates. With these changes, Apple devices may be used for high-quality mobile audio recording.

Note: The 5G iPod, 2G nano, iPod classic, and 3G nano support both stereo and monophonic microphones through the 30-pin connector. The 4G iPod only supports monophonic microphones plugged into the 9-pin audio/remote connector; it cannot record audio from a microphone connected to the 30-pin connector. See "The 9-Pin Audio/Remote Connector" in the chapter "Historical Information" in *MFi Accessory Hardware Specification*.

The Microphone lingo is defined such that the Apple device initiates commands and the accessory responds to these commands; that is, the Apple device sends commands to the accessory and the accessory responds with data or `AccessoryAck` commands.

When the Apple device detects an accessory speaking the Microphone lingo, it may transition into a recorder application where it can create and manage recordings. Based on the microphone accessory capabilities, the Apple device recording application may choose to change its appearance based on the presence or absence of certain microphone features. The accessory should indicate its capabilities to the Apple device on request. These capabilities may include:

- Stereo line input source
- Stereo/mono control
- Recording level control
- Recording level limiter

Note: Accessories using Authentication 2.0 and identifying for the Microphone lingo may have audio routed to them by default, based on their resistor ID, identification method and/or Apple device model. If a microphone accessory does not support line-out (i.e. it does not have a built-in speaker) and it uses IDPS for identification, its `AccessoryCapsToken` must not confirm line-out support (see Table 3-70 (page 163)).

Microphone accessory devices can draw power from the Apple device or supply power to the Apple device. Accessory power management is important as Apple devices transition to a smaller physical size at the same time as trying to extend battery life. An accessory using the Microphone lingo must remain in low power mode, as defined in the section "Accessory Power Policy" in *MFi Accessory Hardware Specification*. It is allowed to transition to high-power mode only if it receives an `iPodModeChange` command with a Mode value of 0x00 or 0x02 (begin audio recording or playback), as listed in Table C-18 (page 537), and also only if it has declared the intermittent high-power option during its identification process. The accessory must return to low power mode within 100 ms of receiving an `iPodModeChange` command with a Mode value of 0x01 or 0x03 (end audio recording or playback). Like all accessories, an accessory using the Microphone lingo must comply with the power requirements of the Apple device's Sleep and Hibernate states, as specified in Appendix A, "Apple Device Power States and Accessory Power," in *MFi Accessory Hardware Specification*. Accessories are informed of Apple device state changes by receiving a General lingo `NotifyiPodStateChange` command.

The microphone accessory is responsible for keeping its power consumption below the maximum allowed limits for each Apple device state; see "Accessory Power Policy" in *MFi Accessory Hardware Specification*. Current from the Apple device on the Accessory Power line of the 30-pin connector is completely shut off when an Apple device enters the Hibernate state. On reset or power up, the accessory must be in low power mode with the amplifier off (that is, with audio input and output disabled).

Microphone state information must be retained locally by the accessory while uninterrupted current from the Apple device (either high or low power) is available. If current from the Apple device is turned off, accessory state information may be lost. Devices are not expected to retain state information across such power down cycles (Hibernate mode).

Level changes in Apple device playback volume may require the accessory to support Display Remote lingo (0x03) functionality.

Table C-12 (page 534) lists the commands available as part of the Microphone lingo.

Note: Legacy Microphone lingo commands 0x00–0x03 are disabled for devices using the 30-pin connector. They are superseded by "Command 0x06: iPodModeChange" (page 536), which returns an `AccessoryAck` command. Deprecated commands 0x00–0x03 are documented in "9-Pin Audio/Remote Connector Commands" (page 528).

Table C-12 Microphone lingo command summary

Command	ID	Data length	Protocol Version	Connector	Authentication Required
`BeginRecord`	0x00	0x00	All	9-pin Audio/Remote	No
`EndRecord`	0x01	0x00	All	9-pin Audio/Remote	No
`BeginPlayback`	0x02	0x00	All	9-pin Audio/Remote	No
`EndPlayback`	0x03	0x00	All	9-pin Audio/Remote	No
`AccessoryAck`	0x04	0x02	1.01	30-pin	Yes
`GetAccessoryAck`	0x05	0x00	1.01	30-pin	Yes
`iPodModeChange`	0x06	0x01	1.01	30-pin	Yes
`GetAccessoryCaps`	0x07	0x00	1.01	30-pin	Yes
`RetAccessoryCaps`	0x08	0x04	1.01	30-pin	Yes
`GetAccessoryCtrl`	0x09	0x00	1.01	30-pin	Yes
`RetAccessoryCtrl`	0x0A	0x02	1.01	30-pin	Yes
`SetAccessoryCtrl`	0x0B	0x02	1.01	30-pin	Yes
Reserved	0x0C–0xFF	N/A	N/A	N/A	N/A

Table C-13 Select Microphone lingo command timings

Command	Timeout	Tries
`GetAccessoryAck` (0x05)	200 ms	1
`GetAccessoryCaps` (0x07)	200 ms	1
`GetAccessoryCtrl` (0x09)	200 ms	1
`SetAccessoryCtrl` (0x0B)	200 ms	1

C.5.1 Command History of the Microphone Lingo

Table C-14 (page 535) shows the history of command changes in the Microphone lingo.

Table C-14 Microphone lingo command history

Lingo version	Command changes	Features
No version	Add: 0x00–0x03	Microphone begin/end record/playback notification commands, 9-pin Audio/Remote connector only
1.00	None	Version number available through `RequestLingoProtocol-Version`
1.01	Add: 0x04–0x0B	`AccessoryAck`, mode change, capabilities, and control support (30-pin connector only)

The commands described in the following sections are used only with the 30-pin connector. For information about 9-pin connector commands, see "9-Pin Audio/Remote Connector Commands" (page 528).

Note: Devices must return responses to Apple device commands in the order in which the commands were received and within the specified time limits. Failing to send responses in the order commands were received or exceeding the command timeout limits could cause a communications failure and result in the accessory being considered not present by the Apple device.

C.5.2 Command 0x04: AccessoryAck

Lingo: 0x01 — Origin: Accessory

The microphone accessory sends this command in response to a command sent from the Apple device. Note that the commands 0x00–0x03 do not require an `AccessoryAck` response. The accessory sends an `AccessoryAck` response when a command that does not return any data has completed, a bad parameter is received, or an unsupported or invalid command is received.

Table C-15 `AccessoryAck` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command status. See Table 3-6 (page 125).
0x*NN*	1	The ID of the command for which the response is being sent.
Packet footer specified in "Command Packets" (page 109).		

C.5.3 Command 0x05: GetAccessoryAck

Lingo: 0x01 — Origin: Apple device

The Apple device sends this command to get an `AccessoryAck` response from a microphone accessory. The Apple device uses this command to "ping" the accessory and determine that it is present and ready to accept commands. In response, the accessory sends the `AccessoryAck` command with command status OK.

The timeout for this command is 200 ms (0.2 second). The accessory must respond within the allotted time; the Apple device will not retry the command. If the accessory does not respond within the specified time, the command will fail and the accessory may be considered not present by the Apple device.

Table C-16 `GetAccessoryAck` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

C.5.4 Command 0x06: iPodModeChange

Lingo: 0x01 — Origin: Apple device

The Apple device sends this command to the microphone accessory when an audio recording or playback event occurs. The microphone accessory uses the `iPodModeChange` command to configure its inputs or outputs and power consumption level for the specified mode. In response, the accessory sends the `AccessoryAck` command with the command status OK. The accessory sends the `AccessoryAck` command when the accessory has completed its mode change.

The Apple device does not wait to receive an `AccessoryAck` command from the accessory in response to the mode change. It may continue sending other commands to the accessory after it has sent the mode change command.

Table C-17 `iPodModeChange` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Mode. See Table C-18 (page 537).
Packet footer specified in "Command Packets" (page 109).		

Table C-18 (page 537) lists the possible values of the Mode byte.

Table C-18 Mode values

Value	Meaning
0x00	Begin audio recording mode. When it receives this command, the accessory can activate its microphone recording inputs or outputs connected to the Apple device's line inputs. If the accessory has requested the intermittent high power option using General lingo "Command 0x13: IdentifyDeviceLingoes" (page 133), it must wait until after this command is received before leaving low power mode, as defined in the section "Accessory Power Policy" in *MFi Accessory Hardware Specification*. By the time the accessory receives this command, accessory high power is enabled and ready to use during the recording process.
0x01	End audio recording mode. When it receives this command, the accessory can deactivate its microphone recording inputs or outputs and return to a quiescent state. If the accessory has requested the intermittent high power option, by using the General lingo command `IdentifyDeviceLingoes`, it may be in a high power consumption state. After receiving this command, the accessory must reduce its power consumption to low power mode, as defined in the section "Accessory Power Policy" in *MFi Accessory Hardware Specification*, within 100 ms.
0x02	Begin audio playback mode. When it receives this command, the accessory can activate its speaker playback inputs or outputs connected to the Apple device's line outputs, if present. If the accessory has requested the intermittent high power option, by using the General lingo command `IdentifyDeviceLingoes`, it must wait until after this command is received before consuming more than low power. By the time the accessory receives this command, accessory high power is enabled and ready to use during the playback process.
0x03	End audio playback mode. When it receives this command, the accessory can deactivate its speaker playback inputs or outputs and return to a quiescent state. If the accessory has requested the intermittent high power option using the General lingo command `IdentifyDeviceLingoes`, it may be in a high power consumption state. After receiving this command, the accessory must reduce its power consumption below low power mode, as defined in the section "Accessory Power Policy" in *MFi Accessory Hardware Specification*, within 100 ms.
0x04–0xFF	Reserved.

Note: Failure to wait for a mode change notification before increasing power consumption could result in an incompatibility with present and future Apple devices.

Failure to reduce power consumption within the stated time could result in an incompatibility with present and future Apple devices.

C.5.5 Command 0x07: GetAccessoryCaps

Lingo: 0x01 — Origin: Apple device

The Apple device sends this command to the microphone accessory to determine the features present on the accessory. In response, the accessory sends "Command 0x08: RetAccessoryCaps" (page 538) with the payload indicating the capabilities it supports.

The timeout for this command is 200 ms (0.2 second). The accessory must respond within the allotted time; the Apple device will not retry the command. If the accessory does not respond within the specified time, the command will fail and the accessory may be considered not present by the Apple device.

Table C-19 `GetAccessoryCaps` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

C.5.6 Command 0x08: RetAccessoryCaps

Lingo: 0x01 — Origin: Accessory

The accessory sends this command in response to the command "Command 0x07: GetAccessoryCaps" (page 537) sent by the Apple device. The microphone accessory returns the payload indicating which capabilities it supports.

Table C-20 `RetAccessoryCaps` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Accessory capabilities . See Table C-21 (page 538).
Packet footer specified in "Command Packets" (page 109).		

The capabilities bit ranges correspond to the microphone control commands. The Apple device should not attempt to control accessory features, using "Command 0x0B: SetAccessoryCtrl" (page 540), if the associated capabilities bits are not set. Table C-21 (page 538) lists the meaning of these bits.

Table C-21 Microphone capabilities bitmask

Bit	Meaning
00	Stereo line input. A value of 0 indicates the accessory is monophonic only.
01	Stereo or mono line input. This bit should be set only if the microphone supports stereo line input and can switch between stereo and mono modes.
02	Recording level is present and variable.
03	Recording level limit is present.

Bit	Meaning
04	Accessory supports duplex audio; it can play audio output from the Apple device while it sends audio input to the Apple device.
31:05	Reserved.

C.5.7 Command 0x09: GetAccessoryCtrl

Lingo: 0x01 — Origin: Apple device

The Apple device sends this command to get the accessory control state for the specified control type. In response, the accessory sends "Command 0x0A: RetAccessoryCtrl" (page 539) with its current control state. If this command is not supported by the accessory—that is, if the microphone does not have any configurable controls—it must return an `AccessoryAck` command with a bad parameter error status.

The timeout for this command is 200 ms (0.2 second). The accessory must respond within the allotted time; the Apple device will not retry the command. If the accessory does not respond within the specified time, the command will fail and the accessory may be considered not present by the Apple device.

Table C-22 `GetAccessoryCtrl` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	The control type for which to get the state. The possible values are: 0x00: Reserved. 0x01: Stereo/mono line input. 0x02: Recording level control. 0x03: Recording level limiter control. 0x04–0xFF: Reserved.
Packet footer specified in "Command Packets" (page 109).		

C.5.8 Command 0x0A: RetAccessoryCtrl

Lingo: 0x01 — Origin: Accessory

The accessory sends this command in response to the command "Command 0x09: GetAccessoryCtrl" (page 539) received from the Apple device. The accessory returns the current control state for the specified control type. Control types are supported only if the associated capabilities bits are set in the command "Command 0x08: RetAccessoryCaps" (page 538).

Table C-23 `RetAccessoryCtrl` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	The control type. See Table C-24 (page 540).
0x*NN*	1	The control data. See Table C-24 (page 540).
Packet footer specified in "Command Packets" (page 109).		

Table C-24 (page 540) lists the different control types and the data associated with them.

Table C-24 Control types and data

Control type value	Control type	Control data
0x00	Reserved	Stereo capability cannot be set.
0x01	Stereo/mono line input	Possible data values are: 0x00 = mono 0x01 = stereo 0x02–0xFF = reserved
0x02	Recording level control	Possible data values are in a range between: 0x00 = mute 0xFF = maximum gain
0x03	Recording level limiter	Possible data values are: 0x00 = off 0x01 = on 0x02–0xFF = reserved
0x04–0xFF	Reserved	

C.5.9 Command 0x0B: SetAccessoryCtrl

Lingo: 0x01 — Origin: Apple device

The Apple device sends this command to set the accessory control state for the specified control type. In response, the accessory sends an `AccessoryAck` command with the command status. If this command is not supported by the accessory—that is, if the microphone does not have any configurable controls—it must return an `AccessoryAck` command with a bad parameter error status.

The timeout for this command is 200 ms (0.2 second). The accessory must respond within the allotted time; the Apple device will not retry the command. If the accessory does not respond within the specified time, the command will fail and the accessory may be considered not present by the Apple device.

Table C-25 `SetAccessoryCtrl` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	The control type. The control type and data values are the same as for the `RetAccessoryCtrl` (0x0A) command. See Table C-24 (page 540) for more information.
0x*NN*	1	The control data. See Table C-24 (page 540) for more information.
Packet footer specified in "Command Packets" (page 109).		

C.5.10 Deprecated MicrophoneCapsToken

Table C-26 `MicrophoneCapsToken` format

Name	Bytes	Value	Description
`length`	1	0x06	Length of this token-value field in bytes, not including this byte.
`FIDType`	1	0x01	First byte of token ID.
`FIDSubtype`	1	0x00	Second byte of token ID.
`micCapsBitmask`	4	0x*NNNNNNNN*	Microphone capabilities; see Table C-27 (page 541).

Table C-27 Microphone capabilities bits

Bit	Meaning
00	Stereo line input. A value of 0 indicates that the accessory is monophonic only.
01	Stereo or mono line input. This bit should be set only if the microphone supports stereo line input and can switch between stereo and mono modes.
02	Recording level is present and variable.
03	Recording level limit is present.
04	Accessory supports duplex audio; it can play audio output from the Apple device while it sends audio input to the Apple device.
31:05	Reserved.

C.6 Deprecated Simple Remote Lingo Command

Note: This command is deprecated; do not use it in new products.

C.6.1 Command 0x02: ImageButtonStatus

Lingo: 0x02 — Origin: Accessory

The accessory sends this command to the Apple device when an image-specific button event occurs. The button status is a bitmask representing each button that is currently pressed. The button status packet must be repeatedly sent by the accessory at intervals between 30 and 100 ms while one or more buttons are pressed. When all buttons are released, the accessory must send a button status packet with a 0x00 payload to indicate that no buttons are pressed. In response, the Apple device will return an `iPodAck` packet containing the command status to the accessory.

Note: This command is not supported in Apple products running iOS 3.2.

Table C-28 `ImageButtonStatus` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Byte index 0, image-specific button states 7:0. See Table C-29 (page 542) for a list of image-specific button states recognized by the Apple device.
0x*NN*	1	Byte index 1, button states 15:8 (optional)
0x*NN*	1	Byte index 2, button states 23:16 (optional)
0x*NN*	1	Byte index 3, button states 31:24 (optional)
Packet footer specified in "Command Packets" (page 109).		

Table C-29 Image-specific button values

Button name	Number	Byte index	Button bitmask
Play/Pause	0	0x00	0x01
Next image	1	0x00	0x02
Previous image	2	0x00	0x04
Stop	3	0x00	0x08
Play/resume	4	0x00	0x10
Pause	5	0x00	0x20

Button name	Number	Byte index	Button bitmask
Shuffle advance	6	0x00	0x40
Repeat advance	7	0x00	0x80
Reserved	15:8	0x01	0xFF
Reserved	23:16	0x02	0xFF
Reserved	31:24	0x03	0xFF

C.7 Deprecated Extended Interface Commands

The commands in this section have been replaced by commands in the iAP General lingo (Lingo 0x00).

C.7.1 Command 0x0012: RequestProtocolVersion

Lingo: 0x04 — Origin: Accessory

Note: This command is deprecated except for use by accessories supporting 3G iPods. Use the General lingo (0x00) command 0x0F, `RequestLingoProtocolVersion`, instead.

Requests the version of the running protocol from the Apple device. The Apple device responds with a "Command 0x0013: ReturnProtocolVersion" (page 543) command.

Note: This command requests the Extended Interface mode protocol version, not the Apple device software version.

Table C-30 `RequestProtocolVersion` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

C.7.2 Command 0x0013: ReturnProtocolVersion

Lingo: 0x04 — Origin: Apple device

Note: This command is deprecated except for use by accessories supporting 3G iPods. Use the General lingo (0x00) command 0x10, `ReturnLingoProtocolVersion`, instead.

Returns the Apple device Extended Interface mode protocol version number. The Apple device sends this command in response to the "Command 0x0012: RequestProtocolVersion" (page 543) command from the accessory. The major version number specifies the protocol version digits to the left of the decimal point; the minor version number specifies the digits to the right of the decimal point. For example, a major version number of 0x01 and a minor version number of 0x08 represents an Extended Interface mode protocol version of 1.08. This protocol information is also available through the General lingo (lingo 0x00) command `RequestLingoProtocolVersion` when passing lingo 0x04 as the lingo parameter.

Note: This command returns the Extended Interface mode protocol version, not the Apple device software version.

Table C-31 `ReturnProtocolVersion` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Protocol major version number
0x*NN*	1	Protocol minor version number
Packet footer specified in "Command Packets" (page 109).		

C.7.3 Command 0x0014: RequestiPodName

Lingo: 0x04 — Origin: Accessory

Note: This command is deprecated except for use by accessories supporting 3G iPods. Use the General lingo (0x00) command 0x07, `RequestiPodName`, instead.

Returns the name of the user's Apple device or "iPod" if the Apple device name is undefined. This allows the Apple device name to be shown in the human-machine interface (HMI) of the interfacing body. The Apple device responds with the "Command 0x0015: ReturniPodName" (page 545) command containing the Apple device name text string.

Table C-32 `RequestiPodName` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

C.7.4 Command 0x0015: ReturniPodName

Lingo: 0x04 — Origin: Apple device

Note: This command is deprecated except for use by accessories supporting 3G iPods. Use the General lingo (0x00) command 0x08, `ReturniPodName`, instead.

The Apple device sends this command in response to the "Command 0x0014: RequestiPodName" (page 544) command from the accessory. The Apple device name is encoded as a null-terminated UTF-8 character array. The Apple device name string is not limited to 252 characters; it may be sent in small or large packet format. The small packet format is shown.

Note: Starting with version 1.07 of the Extended Interface lingo, the `ReturniPodName` command on Windows-formatted Apple devices returns the iTunes name of the Apple device instead of the Windows volume name.

Table C-33 `ReturniPodName` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	*NN*	Apple device name as UTF-8 character array
Packet footer specified in "Command Packets" (page 109).		

C.7.5 Command 0x0028: PlayCurrentSelection

Lingo: 0x04 — Origin: Accessory

Applies To: Playback and Database Engines. This command copies items from the database engine to the Playback Engine.

Note: This command is **deprecated** for new accessory designs. Use `SelectDBRecord` instead.

Requests playback of the currently selected track or list of tracks. The currently selected tracks are first placed in the Now Playing playlist. If the Apple device is set to shuffle, the tracks are shuffled. Then the track record index is passed to the player to play, as described in "Playback Behavior" (page 58). The Apple device also returns an an `iPodAck` command indicating the status of the `PlayCurrentSelection` command.

Note: Any track that has the Skip When Shuffle attribute will be excluded from the Now Playing playlist unless its track record index is passed by this command, in which case it will play. With a playlist that contains only skip-when-shuffle tracks, the next action depends on the Apple device model. With iOS devices, the next track will be shuffled and played; with other Apple devices, no further tracks will play.

Table C-34 `PlayCurrentSelection` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	4	Selection track record index
Packet footer specified in "Command Packets" (page 109).		

C.7.6 Command 0x0038: SelectSortDBRecord

Lingo: 0x04 — Origin: Accessory

Applies To: Database Engine

Note: This command is **deprecated.** Use `SelectDBRecord` instead.

This command acts the same as "Command 0x0017: SelectDBRecord" (page 415), but includes a sorting feature. It selects one or more records in the Apple device database, based on a category-relative index. For example, selecting category 2 (Artist), record index 1, and sort order 3 (Album) results in a list of selected tracks (records) from the second artist in the artist list, sorted by album name. Selections are additive and limited by the category hierarchy; see "Database Category Hierarchies" (page 60). Subsequent selections are made based on the subset of records resulting from previous selections and not from the entire database.

Table C-35 `SelectSortDBRecord` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Database category type. See Table 5-27 (page 416).
0x*NN*	4	Category record index
0x*NN*	1	Database sort type. See Table C-36 (page 547).
Packet footer specified in "Command Packets" (page 109).		

Table C-36 (page 547) shows the possible sort orders.

Table C-36 Database sort order options

Sort Order	Code	Protocol version
Sort by genre	0x00	1.00
Sort by artist	0x01	1.00
Sort by composer	0x02	1.00
Sort by album	0x03	1.00
Sort by name	0x04	1.00
Reserved	0x05	N/A
Sort by release date	0x06	1.08
Sort by series (video only)	0x07	1.12
Sort by season (video only)	0x08	1.12
Sort by episode (video only)	0x09	1.12
Reserved	0x0A – 0xFE	N/A
Use default sort type	0xFF	1.00

C.8 Deprecated Lingo 0x05: Accessory Power Lingo

Note: This iAP lingo is deprecated; do not use it in new products.

The Accessory Power lingo was used by devices that draw up to 100 mA peak current from the Apple device through pin 13 of the 30-pin connector (Accessory Power). To use this lingo, accessories must identify themselves as intermittent high-power devices during the identification process. They may include accessories that transmit an Apple device's analog audio over radio frequencies, typically over an unused frequency in the FM band.

Note: Any accessory drawing power from an Apple device must conform to the requirements of the section "Accessory Power Policy" in *MFi Accessory Hardware Specification*.

The Accessory Power lingo is intended for use in conjunction with audio playback from the Apple device. The accessory must remain in low power mode, as defined in the section "Accessory Power Policy" in *MFi Accessory Hardware Specification*, until it receives a `BeginHighPower` command, which notifies it that it may begin drawing high power. The `EndHighPower` command notifies the accessory that it must stop drawing high power and return to low power mode within 1 second of receiving the command.

Like all accessories, an accessory using the Accessory Power lingo must comply with the power requirements of the Apple device's Sleep and Hibernate states, as specified in "Apple Device Power States" in *MFi Accessory Hardware Specification*. Accessories are informed of Apple device state changes by receiving a General lingo `NotifyiPodStateChange` command.

Table C-37 Accessory Power lingo command summary

Command	ID	Data length	Protocol version	Authentication required
Reserved	0x00–0x01	N/A	N/A	N/A
`BeginHighPower`	0x02	0x00	All	UART: No USB: Yes
`EndHighPower`	0x03	0x00	All	
Reserved	0x04–0xFF	N/A	N/A	N/A

C.8.1 Command History of the Accessory Power lingo

Table C-38 (page 548) shows the history of command changes in the Accessory Power lingo:

Table C-38 Accessory Power lingo command history

Lingo version	Command changes	Features
(0.00)	Add: 0x02–0x03	`Begin/EndHighPower` support
1.00	None	Version number available through `RequestLingoProtocol-Version`
1.01	None	BugFix: `BeginTransmission` command sent after accessory inserted while the Apple device is playing

C.8.2 Command 0x02: BeginHighPower

Lingo: 0x05 — Origin: Apple device

The Apple device sends this command to notify the accessory that high power may be used. Upon receipt of the this command, the accessory may begin drawing more than low power, up to a maximum of 100 mA, if it has previously requested intermittent high power during its identification process.

Table C-39 `BeginHighPower` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

C.8.3 Command 0x03: EndHighPower

Lingo: 0x05 — Origin: Apple device

The Apple device sends this command to notify the accessory to stop using accessory high power. The accessory must reduce its power usage to low power within 1 second of receiving an `EndHighPower` packet.

Note: Failure to reduce current drawn from the Apple device to low power status within 1 second after receiving this notification could damage the Apple device.

Table C-40 `EndHighPower` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

C.9 Deprecated USB Host Control Commands

Note: These commands are deprecated; do not use them in new products. For USB Host Mode functionality, see "Lingo 0x06: USB Host Mode Lingo" (page 283).

C.9.1 Command 0x00: AccessoryAck

Lingo: 0x06 — Origin: Accessory

This command is sent by the accessory in response to a command received from the Apple device. It reports the original command number and the status of the command.

Table C-41 `AccessoryAck` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Command status; see Table 3-6 (page 125).
0x*NN*	1	ID of the command being acknowledged
Packet footer specified in "Command Packets" (page 109).		

C.9.2 Command 0x01: GetUSBPowerState

Lingo: 0x06 — Origin: Apple device

This command is sent by the Apple device to obtain the accessory's current USB power state. In response, the accessory must send a `RetUSBPowerState` command with the current USB power setting.

Table C-42 `GetUSBPowerState` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
No parameters.		
Packet footer specified in "Command Packets" (page 109).		

C.9.3 Command 0x02: RetUSBPowerState

Lingo: 0x06 — Origin: Accessory

This command is sent by the accessory in response to a `GetUSBPowerState` command received from the Apple device. It returns the current state of the USB power supply.

Table C-43 `RetUSBPowerState` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Current power state (zero for off, nonzero for on)
Packet footer specified in "Command Packets" (page 109).		

C.9.4 Command 0x03: SetUSBPowerState

Lingo: 0x06 — Origin: Apple device

This command is sent by the Apple device to set the accessory's USB power state. The accessory must set the USB power state and respond with an `AccessoryAck` command indicating command completion.

Table C-44 `SetUSBPowerState` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	1	Power state to be set (zero for off, nonzero for on)
Packet footer specified in "Command Packets" (page 109).		

C.10 Deprecated iPod Out Lingo Command

Note: This command is deprecated; do not use it in new products.

C.10.1 Command 0x06: DevVideoScreenInfo

Lingo: 0x0D — Origin: Accessory

Accessories formerly used this command to dynamically change the screen size they allot to the Apple device. The Apple device acknowledged it with an `iPodAck` command.

Table C-45 `DevVideoScreenInfo` packet

Value	Bytes	Parameter
Packet header specified in "Command Packets" (page 109).		
0x*NN*	2	`totalScreenWidthInches`: Accessory's approximate screen width in inches
0x*NN*	2	`totalScreenHeightInches`: Accessory's approximate screen height in inches
0x*NN*	2	`totalScreenWidthPixels`: Number of pixels along the accessory's screen width
0x*NN*	2	`totalScreenHeightPixels`: Number of pixels along the accessory's screen height
0x*NN*	2	`iPodOutScreenWidthPixels`: Number of pixels along the iPod Out screen width allotment
0x*NN*	2	`iPodOutScreenHeightPixels`: Number of pixels along the iPod Out screen height allotment
0x*NN*	1	`screenFeaturesMask`: A bitmask with bits set to represent the accessory screen's features; see Table C-46 (page 551).
0x*NN*	1	`screenGammaValue`: A representation of the accessory display's gamma value. The Apple device obtains the gamma value from this byte by dividing it by 100 and adding 1.00. For example, a gamma of 2.200 is represented by a `screenGammaValue` of 120.
Packet footer specified in "Command Packets" (page 109).		

Table C-46 Accessory screen features

Bit	Meaning
00	The accessory has a color display
07-01	Reserved

C.11 Authentication 1.0 Sample Command Sequence

For legacy purposes, Table C-47 (page 552) shows the process implemented by some existing accessories, using authentication 1.0. This process is not supported for new accessory designs.

Table C-47 Legacy accessory authentication

Step	Action or command	Direction	Comments
1	`IdentifyDeviceLingoes` (0x13)	Acc to Dev	The accessory identifies itself, lists its supported lingoes, and requests authentication. It can request immediate authentication or it can defer authentication until it tries to use restricted lingoes or commands.
2	`iPodAck` (0x02)	Dev to Acc	The Apple device acknowledges receipt of `IdentifyDeviceLingoes`.
5	`GetAccessory-AuthenticationInfo` (0x14)	Dev to Acc	The Apple device requests accessory authentication information and starts its timeout timer.
6	`RetAccessory-AuthenticationInfo` (0x15)	Acc to Dev	The accessory returns its major and minor authentication version.
7	`AckAccessory-AuthenticationInfo` (0x16)	Dev to Acc	The Apple device returns the status of its check of the authentication version. If it does not support the authentication version, the Apple device sends a value of 0x08 to the accessory.
The Apple device lingoes and commands are enabled after the authentication version is validated.			
8	`GetAccessoryInfo` (0x27)	Dev to Acc	The Apple device queries the accessory for information about it.
9	`RetAccessoryInfo` (0x28)	Acc to Dev	The accessory returns information about its identity and capabilities.
10	`GetAccessory-Authentication-Signature` (0x17)	Dev to Acc	The Apple device sends a 16-byte random challenge to the accessory and asks it to calculate a digital signature.
11	`RetAccessory-Authentication-Signature` (0x18)	Acc to Dev	The accessory returns its digital signature to the Apple device within 7.5 seconds.
12	`AckAccessory-AuthenticationStatus` (0x19)	Dev to Acc	The Apple device verifies the signature by deriving a public key from the Device ID and returns the status of signature verification.

C.12 Accessory Identification With Non-IDPS Apple Devices

Some models of Apple devices do not support IDPS. Every accessory must always start its identification process by sending `StartIDPS`; if the Apple device's `iPodAck` response passes a `status` value of 0x04 (Bad Parameter), the accessory must revert to an identification process using `IdentifyDeviceLingoes`.

Note: The accessory must send `IdentifyDeviceLingoes` within 800 ms of receiving the Apple device's `iPodAck` response.

The sample command listings in this section illustrate various forms of the identification process using `IdentifyDeviceLingoes`:

- Table C-48 (page 553) shows the basic process for accessories that communicate using UART transport.
- Table C-49 (page 555) shows the basic process for accessories that communicate using USB or Bluetooth.
- Table C-50 (page 556) lists sample commands for an accessory that supports the General and Extended Interface lingoes. For Extended Interface mode command details, see "Extended Interface Mode" (page 397).
- Table C-52 (page 565) lists sample commands for an accessory that supports the General, Simple Remote, and Display Remote lingoes.
- Table C-53 (page 568) lists sample commands for an accessory that supports the General, Simple Remote, Display Remote, and Digital Audio lingoes.
- Table C-54 (page 571) shows a sample sequence of commands that implements radio tagging in an accessory. See "iTunes Tagging Accessory Design" (page 467).
- Table C-55 (page 573) shows a sample sequence of commands that implements the Nike + iPod cardio equipment system in an accessory. See "Nike + iPod Cardio Accessory Design" (page 501).

Table C-48 UART accessory identification with non-IDPS Apple devices

Step	Action or command	Direction	Comments
1	Wait 80 ms after the Apple device turns on Accessory Power (pin 13 in "Hardware Interfaces" in *MFi Accessory Hardware Specification*)	Acc	Wait for the Apple device's internal bootstrap and wakeup. If the Apple device is in Sleep mode, the accessory must repeat Steps 1-5 until the Apple device wakes and the accessory receives an `iPodAck` command.
2	Send sync byte (0xFF)	Acc to Dev	Allow the Apple device to synchronize to the accessory's baud rate.
3	Wait 20 ms	Acc	
4	`StartIDPS` (0x38)	Acc to Dev	The accessory sends `StartIDPS` to start the identification process specified in "Identification" (page 93).

Step	Action or command	Direction	Comments
5	Wait up to 1 second	Acc	The accessory waits for the Apple device to send an `iPodAck` command acknowledging receipt of `StartIDPS`.
6	`iPodAck` (0x02) of `StartIDPS`	Dev to Acc	The Apple device sends a status of 0x04 (Bad Parameter) to indicate that it does not support IDPS; see Table 2-4 (page 98).
7	`IdentifyDeviceLingoes` (0x13)	Acc to Dev	The accessory sends `IdentifyDeviceLingoes` with the lingo mask set to only the General lingo, the option bit mask set to 0x0, and the Device ID set to 0x0. This cancels any active authentication process, as detailed in "Canceling a Current Authentication Process With IdentifyDeviceLingoes" (page 135).
8	Wait up to 1 second	Acc	The accessory waits for the Apple device to send an `iPodAck` command acknowledging receipt of `IdentifyDeviceLingoes`.
9	`iPodAck` (0x02) of `IdentifyDeviceLingoes`	Dev to Acc	The Apple device sends a status of 0x00 (OK) to acknowledge receipt of the empty `IdentifyDeviceLingoes` command.
If the Apple device does not send an `iPodAck` command within 1 second, it may be a 3G iPod. In this case, if the accessory needs to support the 3G iPod and it is using UART communication, in may retry Step 7 up to 3 times, for a total elapsed time of 3 seconds. If the accessory does not receive an `iPodAck` command after the third try, it may either quit or send an `Identify` command to achieve limited use of iAP. See "The 3G iPod" (page 527) and "Interfacing With the 3G iPod" in *MFi Accessory Hardware Specification*.			
10	`GetAccessoryInfo` (0x27)	Dev to Acc	The Apple device queries the accessory for information about it. The accessory ignores this query.
11	`IdentifyDeviceLingoes` (0x13)	Acc to Dev	The accessory sends a second `IdentifyDeviceLingoes` command, specifying which lingoes it supports and requesting immediate authentication (see "Authentication" (page 103)).
12	`iPodAck` (0x02) of `IdentifyDeviceLingoes`	Dev to Acc	The Apple device acknowledges receipt of the second `IdentifyDeviceLingoes` command.
13	`RequestTransportMaxPayloadSize` (0x11)	Acc to Dev	The accessory queries the Apple device to determine the maximum packet payload size provided by the current iAP transport; see "Command 0x11: RequestTransportMaxPayloadSize" (page 132).
14	`ReturnTransportMaxPayloadSize` (0x12)	Dev to Acc	The Apple device returns the current transport's maximum payload size.

Step	Action or command	Direction	Comments
The Apple device now initiates the authentication process by sending a GetAccessoryAuthenticationInfo command to the accessory, as shown in Table 2-8 (page 106).			

Table C-49 USB or BT accessory identification with non-IDPS Apple devices

Step	Action or command	Direction	Comments
1	StartIDPS (0x38)	Acc to Dev	The accessory sends StartIDPS to start the identification process specified in "Identification" (page 93).
2	Wait up to 1 second	Acc	The accessory waits for the Apple device to send an iPodAck command acknowledging receipt of StartIDPS.
3	iPodAck (0x02) of StartIDPS	Dev to Acc	The Apple device sends a status of 0x04 (Bad Parameter) to indicate that it does not support IDPS.
4	IdentifyDeviceLingoes (0x13)	Acc to Dev	The accessory sends IdentifyDeviceLingoes with the lingo mask set to only the General lingo, the option bit mask set to 0x0, and the Device ID set to 0x0. This cancels any active authentication process, as detailed in "Canceling a Current Authentication Process With IdentifyDeviceLingoes" (page 135).
5	Wait up to 1 second	Acc	The accessory waits for the Apple device to send an iPodAck command acknowledging receipt of IdentifyDeviceLingoes.
6	iPodAck (0x02) of IdentifyDeviceLingoes	Dev to Acc	The Apple device sends a status of 0x00 (OK) to acknowledge receipt of the empty IdentifyDeviceLingoes command.
7	GetAccessoryInfo (0x27)	Dev to Acc	The Apple device queries the accessory for information about it. The accessory ignores this query.
8	IdentifyDeviceLingoes (0x13)	Acc to Dev	The accessory sends a second IdentifyDeviceLingoes command, specifying which lingoes it supports and requesting immediate authentication (see "Authentication" (page 103)).
9	iPodAck (0x02) of IdentifyDeviceLingoes	Dev to Acc	The Apple device acknowledges receipt of the second IdentifyDeviceLingoes command.
10	RequestTransportMaxPayloadSize (0x11)	Acc to Dev	The accessory queries the Apple device to determine the maximum packet payload size provided by the current iAP transport; see "Command 0x11: RequestTransportMaxPayloadSize" (page 132).

Step	Action or command	Direction	Comments
11	`ReturnTransport-MaxPayloadSize` (0x12)	Dev to Acc	The Apple device returns the current transport's maximum payload size.
The Apple device now initiates the authentication process by sending a `GetAccessoryAuthenticationInfo` command to the accessory, as shown in Table 2-8 (page 106).			

Table C-50 Non-IDPS identification of lingoes 0x00+0x04

Step	Accessory command	Apple device command	Comment
Steps A–C cancel any current authentication process; see "Canceling a Current Authentication Process With IdentifyDeviceLingoes" (page 135). Step B also detects if the attached Apple device is a 3G iPod.			
A	`IdentifyDevice-Lingoes`		identifying lingo 'General'; options none; device ID 0x00000000
The accessory waits up to 1 second for the Apple device to respond.			
B		`iPodAck`	acknowledging 'Success (OK)' to command 'General Lingo::IdentifyDeviceLingoes'
If the Apple device does not send an `iPodAck` command within 1 second, it may be a 3G iPod. In this case, if the accessory needs to support the 3G iPod and it is using UART communication, in may retry Step A up to 3 times, for a total elapsed time of 3 seconds. If the accessory does not receive an `iPodAck` command after the third try, it may either quit or send an `Identify` command to achieve limited use of iAP. See "The 3G iPod" (page 527) and "Interfacing With the 3G iPod" in *MFi Accessory Hardware Specification*.			
C		`GetAccessoryInfo`	querying the accessory; accessory info type = 0x00 (get accessory info capabilities). The accessory ignores this query.
D	`RequestTransport-MaxPayloadSize`		requesting transport maximum payload size
E		`ReturnTransport-MaxPayloadSize`	returning transport maximum payload size
The accessory now proceeds to identify its lingoes to the attached Apple device.			
1	`IdentifyDevice-Lingoes`		identifying lingoes 'General|Extended Interface'; options 'auth:immediate; power:low; device ID 0x00000200
2		`iPodAck`	acknowledging 'Success (OK)' to command 'General Lingo::IdentifyDeviceLingoes'
3		`GetAccessory-AuthenticationInfo`	no params

Step	Accessory command	Apple device command	Comment
4	RetAccessory-AuthenticationInfo		returning auth protocol v2.0; section: 0/1;
5		iPodAck	acknowledging 'Success (OK)' to command 'General Lingo::RetAccessoryAuthenticationInfo'
6	RetAccessory-AuthenticationInfo		returning auth protocol v2.0; section: 1/1;
7		AckAccessory-AuthenticationInfo	acknowledging 'auth info supported'
8		GetAccessoryInfo	requesting 'Acc info capabilities'
9		GetAccessory-Authentication-Signature	offering challenge
10	RetAccessoryInfo		returning 'Acc info capabilities|Acc name|Acc FW version|Acc HW version|Acc manufacturer|Acc model number|Acc serial number|Acc incoming max payload size' in response to 'Acc info capabilities'
11		GetAccessoryInfo	requesting 'Acc name'
12	RetAccessory-Authentication-Signature		returning signature
13	RetAccessoryInfo		returning 'Apple' in response to 'Acc name'
14		AckAccessory-AuthenticationStatus	acknowledging authentication status 'Success (OK)'
15		GetAccessoryInfo	requesting 'Acc FW version'
16	RetAccessoryInfo		returning 'v9.8.7' in response to 'Acc FW version'
17		GetAccessoryInfo	requesting 'Acc HW version'
18	RetAccessoryInfo		returning 'v1.2.3' in response to 'Acc HW version'
19		GetAccessoryInfo	requesting 'Acc manufacturer'
20	RetAccessoryInfo		returning 'Apple Inc.' in response to 'Acc manufacturer'
21		GetAccessoryInfo	requesting 'Acc model number'

Step	Accessory command	Apple device command	Comment
22	RetAccessoryInfo		returning 'ModelNumber-XYZ' in response to 'Acc model number'
23		GetAccessoryInfo	requesting 'Acc serial number'
24	RetAccessoryInfo		returning 'SerialNumber-1234' in response to 'Acc serial number'
25		GetAccessoryInfo	requesting 'Acc incoming max payload size'
26	RetAccessoryInfo		returning '1024 bytes' in response to 'Acc incoming max payload size'
27	EnterExtended-InterfaceMode		no params
28		iPodAck	acknowledging 'Command Pending (max wait time: 3000 ms)' to command 'General Lingo::EnterExtendedInterfaceMode'
29		iPodAck	acknowledging 'Success (OK)' to command 'General Lingo::EnterExtendedInterfaceMode'
30	RequestExtended-InterfaceMode		no params
31		ReturnExtended-InterfaceMode	returning 'Extended Interface Mode'
32	ResetDBSelection		no params
33		iPodAck	acknowledging 'Success (OK)' to command 'Extended Interface Lingo::ResetDBSelection'
34	GetNumberCategorized-DBRecords		for category 'Playlist'
35		ReturnNumber-CategorizedDBRecords	returning 'record count 22'
36	RetrieveCategorized-DatabaseRecords		requesting 'category Playlist|record start index 0|record read count 4'
37		ReturnCategorized-DatabaseRecord	returning 'record category index 0|MyiPhone
38		ReturnCategorized-DatabaseRecord	returning 'record category index 1|Purchased

Step	Accessory command	Apple device command	Comment
39		ReturnCategorized-DatabaseRecord	returning 'record category index 2|Against Doctor's Orders
40		ReturnCategorized-DatabaseRecord	returning 'record category index 3|Always In Your Mind
41	SelectDBRecord		selecting 'type Playlist|record index 3
42		iPodAck	acknowledging 'Success (OK)' to command 'Extended Interface Lingo::SelectDBRecord'
43	PlayCurrentSelection		playing selection track index 0
44		iPodAck	acknowledging 'Success (OK)' to command 'Extended Interface Lingo::PlayCurrentSelection'
45	SetPlayStatusChange-Notification		setting 'basic play state changes|track index'
46		iPodAck	acknowledging 'Success (OK)' to command 'Extended Interface Lingo::SetPlayStatusChangeNotification'
47	PlayControl		sending 'Toggle Play/Pause'
48		iPodAck	acknowledging 'Success (OK)' to command 'Extended Interface Lingo::PlayControl'
49	PlayControl		sending 'Next Track'
50		iPodAck	acknowledging 'Success (OK)' to command 'Extended Interface Lingo::PlayControl'
51		PlayStatusChange-Notification	notifying new play status 'playback track changed' (new track record index 1)
52	PlayControl		sending 'Toggle Play/Pause'
53		iPodAck	acknowledging 'Success (OK)' to command 'Extended Interface Lingo::PlayControl'
54	SetCurrentPlaying-Track		setting currently playing track to 3
55		iPodAck	acknowledging 'Success (OK)' to command 'Extended Interface Lingo::SetCurrentPlayingTrack'

Step	Accessory command	Apple device command	Comment
56		PlayStatusChange-Notification	notifying new play status 'playback track changed' (new track record index 3)
57	GetPlayStatus		no params
58		ReturnPlayStatus	returning 'Playing|track length 155506 ms|track position 15637 ms'
59	GetIndexedPlaying-TrackTitle		requesting title name for track index 3
60		ReturnIndexedPlaying-TrackTitle	returning Auld Lang Syne
61	GetIndexedPlaying-TrackArtistName		requesting artist name for track index 3
62		ReturnIndexedPlaying-TrackArtistName	returning Straight No Chaser
63	GetIndexedPlaying-TrackAlbumName		requesting album name for track index 3
64		ReturnIndexedPlaying-TrackAlbumName	returning Holiday Spirits (Bonus Track Version)
65	ExitExtended-InterfaceMode		no params
66		iPodAck	acknowledging 'Command Pending (max wait time: 3000 ms)' to command 'General Lingo::ExitExtendedInterfaceMode'
67		iPodAck	acknowledging 'Success (OK)' to command 'General Lingo::ExitExtendedInterfaceMode'

Table C-51 Non-IDPS identification of lingoes 0x00+0x02+0x03

Step	Accessory command	Apple device command	Comment
Steps A–C cancel any current authentication process; see "Canceling a Current Authentication Process With IdentifyDeviceLingoes" (page 135). Step B also detects if the attached Apple device is a 3G iPod.			
A	IdentifyDevice-Lingoes		identifying lingo 'General'; options none; device ID 0x00000000
The accessory waits up to 1 second for the Apple device to respond.			

Step	Accessory command	Apple device command	Comment
B		iPodAck	acknowledging 'Success (OK)' to command 'General Lingo::IdentifyDeviceLingoes'
If the Apple device does not send an iPodAck command within 1 second, it may be a 3G iPod. In this case, if the accessory needs to support the 3G iPod and it is using UART communication, in may retry Step A up to 3 times, for a total elapsed time of 3 seconds. If the accessory does not receive an iPodAck command after the third try, it may either quit or send an Identify command to achieve limited use of iAP. See "The 3G iPod" (page 527) and "Interfacing With the 3G iPod" in *MFi Accessory Hardware Specification*.			
C		GetAccessoryInfo	querying the accessory; accessory info type = 0x00 (get accessory info capabilities). The accessory ignores this query.
D	RequestTransport-MaxPayloadSize		requesting transport maximum payload size
E		ReturnTransport-MaxPayloadSize	returning transport maximum payload size
The accessory now proceeds to identify its lingoes to the attached Apple device.			
1	IdentifyDevice-Lingoes		identifying lingoes 'General|Extended Interface'; options 'auth:immediate; power:low; device ID 0x00000200
2		iPodAck	acknowledging 'Success (OK)' to command 'General Lingo::IdentifyDeviceLingoes'
3		GetAccessory-AuthenticationInfo	no params
4	RetAccessory-AuthenticationInfo		returning auth protocol v2.0; section: 0/1;
5		iPodAck	acknowledging 'Success (OK)' to command 'General Lingo::RetAccessoryAuthenticationInfo'
6	RetAccessory-AuthenticationInfo		returning auth protocol v2.0; section: 1/1;
7		AckAccessory-AuthenticationInfo	acknowledging 'auth info supported'
8		GetAccessoryInfo	requesting 'Acc info capabilities'
9		GetAccessory-Authentication-Signature	offering challenge

Step	Accessory command	Apple device command	Comment
10	RetAccessoryInfo		returning 'Acc info capabilities|Acc name|Acc FW version|Acc HW version|Acc manufacturer|Acc model number|Acc serial number|Acc incoming max payload size' in response to 'Acc info capabilities'
11		GetAccessoryInfo	requesting 'Acc name'
12	RetAccessory-Authentication-Signature		returning signature
13	RetAccessoryInfo		returning 'Apple' in response to 'Acc name'
14		AckAccessory-AuthenticationStatus	acknowledging authentication status 'Success (OK)'
15		GetAccessoryInfo	requesting 'Acc FW version'
16	RetAccessoryInfo		returning 'v9.8.7' in response to 'Acc FW version'
17		GetAccessoryInfo	requesting 'Acc HW version'
18	RetAccessoryInfo		returning 'v1.2.3' in response to 'Acc HW version'
19		GetAccessoryInfo	requesting 'Acc manufacturer'
20	RetAccessoryInfo		returning 'Apple Inc.' in response to 'Acc manufacturer'
21		GetAccessoryInfo	requesting 'Acc model number'
22	RetAccessoryInfo		returning 'ModelNumber-XYZ' in response to 'Acc model number'
23		GetAccessoryInfo	requesting 'Acc serial number'
24	RetAccessoryInfo		returning 'SerialNumber-1234' in response to 'Acc serial number'
25		GetAccessoryInfo	requesting 'Acc incoming max payload size'
26	RetAccessoryInfo		returning '1024 bytes' in response to 'Acc incoming max payload size'
27	EnterExtended-InterfaceMode		no params

Step	Accessory command	Apple device command	Comment
28		iPodAck	acknowledging 'Command Pending (max wait time: 3000 ms)' to command 'General Lingo::EnterExtendedInterfaceMode'
29		iPodAck	acknowledging 'Success (OK)' to command 'General Lingo::EnterExtendedInterfaceMode'
30	RequestExtended-InterfaceMode		no params
31		ReturnExtended-InterfaceMode	returning 'Extended Interface Mode'
32	ResetDBSelection		no params
33		iPodAck	acknowledging 'Success (OK)' to command 'Extended Interface Lingo::ResetDBSelection'
34	GetNumberCategorized-DBRecords		for category 'Playlist'
35		ReturnNumber-CategorizedDBRecords	returning 'record count 22'
36	RetrieveCategorized-DatabaseRecords		requesting 'category Playlist|record start index 0|record read count 4'
37		ReturnCategorized-DatabaseRecord	returning 'record category index 0|MyiPhone
38		ReturnCategorized-DatabaseRecord	returning 'record category index 1|Purchased
39		ReturnCategorized-DatabaseRecord	returning 'record category index 2|Against Doctor's Orders
40		ReturnCategorized-DatabaseRecord	returning 'record category index 3|Always In Your Mind
41	SelectDBRecord		selecting 'type Playlist|record index 3
42		iPodAck	acknowledging 'Success (OK)' to command 'Extended Interface Lingo::SelectDBRecord'
43	PlayCurrentSelection		playing selection track index 0
44		iPodAck	acknowledging 'Success (OK)' to command 'Extended Interface Lingo::PlayCurrentSelection'

Step	Accessory command	Apple device command	Comment
45	SetPlayStatusChange-Notification		setting 'basic play state changes|track index'
46		iPodAck	acknowledging 'Success (OK)' to command 'Extended Interface Lingo::SetPlayStatusChangeNotification'
47	PlayControl		sending 'Toggle Play/Pause'
48		iPodAck	acknowledging 'Success (OK)' to command 'Extended Interface Lingo::PlayControl'
49	PlayControl		sending 'Next Track'
50		iPodAck	acknowledging 'Success (OK)' to command 'Extended Interface Lingo::PlayControl'
51		PlayStatusChange-Notification	notifying new play status 'playback track changed' (new track record index 1)
52	PlayControl		sending 'Toggle Play/Pause'
53		iPodAck	acknowledging 'Success (OK)' to command 'Extended Interface Lingo::PlayControl'
54	SetCurrentPlaying-Track		setting currently playing track to 3
55		iPodAck	acknowledging 'Success (OK)' to command 'Extended Interface Lingo::SetCurrentPlayingTrack'
56		PlayStatusChange-Notification	notifying new play status 'playback track changed' (new track record index 3)
57	GetPlayStatus		no params
58		ReturnPlayStatus	returning 'Playing|track length 155506 ms|track position 15637 ms'
59	GetIndexedPlaying-TrackTitle		requesting title name for track index 3
60		ReturnIndexedPlaying-TrackTitle	returning Auld Lang Syne
61	GetIndexedPlaying-TrackArtistName		requesting artist name for track index 3

Step	Accessory command	Apple device command	Comment
62		ReturnIndexedPlaying-TrackArtistName	returning Straight No Chaser
63	GetIndexedPlaying-TrackAlbumName		requesting album name for track index 3
64		ReturnIndexedPlaying-TrackAlbumName	returning Holiday Spirits (Bonus Track Version)
65	ExitExtended-InterfaceMode		no params
66		iPodAck	acknowledging 'Command Pending (max wait time: 3000 ms)' to command 'General Lingo::ExitExtendedInterfaceMode'
67		iPodAck	acknowledging 'Success (OK)' to command 'General Lingo::ExitExtendedInterfaceMode'

Table C-52 Non-IDPS identification of lingoes 0x00+0x02+0x03

Step	Accessory command	Apple device command	Comment
Steps A–C cancel any current authentication process; see "Canceling a Current Authentication Process With IdentifyDeviceLingoes" (page 135). Step B also detects if the attached Apple device is a 3G iPod.			
A	IdentifyDevice-Lingoes		identifying lingo 'General'; options none; device ID 0x00000000
The accessory waits up to 1 second for the Apple device to respond.			
B		ACK	acknowledging 'Success (OK)' to command 'General Lingo::IdentifyDeviceLingoes'
If the Apple device does not send an ACK command within 1 second, it may be a 3G iPod. In this case, if the accessory needs to support the 3G iPod and it is using UART communication, in may retry Step A up to 3 times, for a total elapsed time of 3 seconds. If the accessory does not receive an ACK command after the third try, it may either quit or send an Identify command to achieve limited use of iAP. See "The 3G iPod" (page 527) and "Interfacing With the 3G iPod" in *MFi Accessory Hardware Specification*.			
C		GetAccessoryInfo	the accessory ignores this query
D	RequestTransport-MaxPayloadSize		requesting transport maximum payload size
E		ReturnTransport-MaxPayloadSize	returning transport maximum payload size
The accessory now proceeds to identify its lingoes to the attached Apple device.			

Step	Accessory command	Apple device command	Comment
1	IdentifyDevice-Lingoes		identifying lingoes 'General|Simple Remote|Display Remote'; options 'auth:immediate; power:low; device ID 0x00000200
2		ACK	acknowledging 'Success (OK)' to command 'General Lingo::IdentifyDeviceLingoes'
3		GetAccessory-AuthenticationInfo	no params
4	RetAccessory-AuthenticationInfo		returning auth protocol v2.0; section: 0/1;
5		ACK	acknowledging 'Success (OK)' to command 'General Lingo::RetAccessoryAuthenticationInfo'
6	RetAccessory-AuthenticationInfo		returning auth protocol v2.0; section: 1/1;
7		AckDevAuthentication-Info	acknowledging 'auth info supported'
8		GetAccessoryInfo	requesting 'Acc info capabilities'
9		GetAccessory-Authentication-Signature	offering challenge
10	RetAccessoryInfo		returning 'Acc info capabilities|Acc name|Acc FW version|Acc HW version|Acc manufacturer|Acc model number|Acc serial number|Acc incoming max payload size' in response to 'Acc info capabilities'
11		GetAccessoryInfo	requesting 'Acc name'
12	RetAccessory-Authentication-Signature		returning signature
13	RetAccessoryInfo		returning 'Apple' in response to 'Acc name'
14		AckDevAuthentication-Status	acknowledging authentication status 'Success (OK)'
15		GetAccessoryInfo	requesting 'Acc FW version'

Step	Accessory command	Apple device command	Comment
16	RetAccessoryInfo		returning 'v9.8.7' in response to 'Acc FW version'
17		GetAccessoryInfo	requesting 'Acc HW version'
18	RetAccessoryInfo		returning 'v1.2.3' in response to 'Acc HW version'
19		GetAccessoryInfo	requesting 'Acc manufacturer'
20	RetAccessoryInfo		returning 'Apple Inc.' in response to 'Acc manufacturer'
21		GetAccessoryInfo	requesting 'Acc model number'
22	RetAccessoryInfo		returning 'ModelNumber-XYZ' in response to 'Acc model number'
23		GetAccessoryInfo	requesting 'Acc serial number'
24	RetAccessoryInfo		returning 'SerialNumber-1234' in response to 'Acc serial number'
25		GetAccessoryInfo	requesting 'Acc incoming max payload size'
26	RetAccessoryInfo		returning '1024 bytes' in response to 'Acc incoming max payload size'
27	SetRemoteEvent-Notification		setting 'Track playback index|Play status'
28		ACK	acknowledging 'Success (OK)' to command 'Display Remote Lingo::SetRemoteEventNotification'
29	GetIndexedPlaying-TrackInfo		getting info type 'Track title' for track index 0 and chapter index 0
30		RetIndexedPlaying-TrackInfo	returning 'Always In Your Mind' for info type 'Track title'
31	GetPlayStatus		no params
32		RetPlayStatus	returning 'Playback paused|index 3|length 155506 ms|position 84933 ms'
33	ContextButtonStatus		sending status 'Play/Pause'
34	ContextButtonStatus		sending status 'All buttons up'
35		RemoteEvent-Notification	reporting status 'Playing' for parameter 'Play status'

Step	Accessory command	Apple device command	Comment
36	GetPlayStatus		no params
37		RetPlayStatus	returning 'Playing|index 3|length 155506 ms|position 91644 ms'
38	ContextButtonStatus		sending status 'Next Track'
39	ContextButtonStatus		sending status 'All buttons up'
40		RemoteEvent-Notification	reporting status 'index 4' for parameter 'Track playback index'
41	SetCurrentPlaying-Track		setting index 0
42		ACK	acknowledging 'Success (OK)' to command 'Display Remote Lingo::SetCurrentPlayingTrack'
43		RemoteEvent-Notification	reporting status 'index 0' for parameter 'Track playback index'
44	GetIndexedPlaying-TrackInfo		getting info type 'Track title' for track index 0 and chapter index 0
45		RetIndexedPlaying-TrackInfo	returning 'Always In Your Mind' for info type 'Track title'

Table C-53 Non-IDPS identification of lingoes 0x00+0x02+0x03+0x0A

Step	Accessory command	Apple device command	Comment
Steps A–C cancel any current authentication process; see "Canceling a Current Authentication Process With IdentifyDeviceLingoes" (page 135). Step B also detects if the attached Apple device is a 3G iPod.			
A	IdentifyDevice-Lingoes		identifying lingo 'General'; options none; device ID 0x00000000
The accessory waits up to 1 second for the Apple device to respond.			
B		ACK	acknowledging 'Success (OK)' to command 'General Lingo::IdentifyDeviceLingoes'
If the Apple device does not send an ACK command within 1 second, it may be a 3G iPod. In this case, if the accessory needs to support the 3G iPod and it is using UART communication, in may retry Step A up to 3 times, for a total elapsed time of 3 seconds. If the accessory does not receive an ACK command after the third try, it may either quit or send an Identify command to achieve limited use of iAP. See "The 3G iPod" (page 527) and "Interfacing With the 3G iPod" in *MFi Accessory Hardware Specification*.			
C		GetAccessoryInfo	the accessory ignores this query

Step	Accessory command	Apple device command	Comment
D	RequestTransport-MaxPayloadSize		requesting transport maximum payload size
E		ReturnTransport-MaxPayloadSize	returning transport maximum payload size
The accessory now proceeds to identify its lingoes to the attached Apple device.			
1	IdentifyDevice-Lingoes		identifying lingoes 'General| Simple Remote| Display Remote| Digital Audio'; options 'auth:immediate; power:low; device ID 0x00000200
2		ACK	acknowledging 'Success (OK)' to command 'General Lingo::IdentifyDeviceLingoes'
3		GetAccessory-AuthenticationInfo	no params
4	RetAccessory-AuthenticationInfo		returning auth protocol v2.0; section: 0/1;
5		ACK	acknowledging 'Success (OK)' to command 'General Lingo::RetAccessoryAuthenticationInfo'
6	RetAccessory-AuthenticationInfo		returning auth protocol v2.0; section: 1/1;
7		AckDevAuthentication-Info	acknowledging 'auth info supported'
8		GetAccSampleRateCaps	no params
9		GetAccessoryInfo	requesting 'Acc info capabilities'
10		GetAccessory-Authentication-Signature	offering challenge
11	RetAccSampleRateCaps		returning sample rates '8000| 11025| 12000| 16000| 22050| 24000| 32000| 44100| 48000'
12	RetAccessoryInfo		returning 'Acc info capabilities| Acc name| Acc FW version| Acc HW version| Acc manufacturer| Acc model number| Acc serial number| Acc incoming max payload size' in response to 'Acc info capabilities'

Step	Accessory command	Apple device command	Comment
13	RetAccessory-Authentication-Signature		returning signature
14		GetAccessoryInfo	requesting 'Acc name'
15	RetAccessoryInfo		returning 'Apple' in response to 'Acc name'
16		TrackNewAudio-Attributes	sample rate 44100; Sound Check value 0; track volume adjustment 0
17	AccAck		acknowledging 'Success (OK)' to command 'Digital Audio Lingo::TrackNewAudioAttributes'
18		AckDevAuthentication-Status	acknowledging authentication status 'Success (OK)'
19		GetAccessoryInfo	requesting 'Acc FW version'
20	RetAccessoryInfo		returning 'v9.8.7' in response to 'Acc FW version'
21		GetAccessoryInfo	requesting 'Acc HW version'
22	RetAccessoryInfo		returning 'v1.2.3' in response to 'Acc HW version'
23		GetAccessoryInfo	requesting 'Acc manufacturer'
24	RetAccessoryInfo		returning 'Apple Inc.' in response to 'Acc manufacturer'
25		GetAccessoryInfo	requesting 'Acc model number'
26	RetAccessoryInfo		returning 'ModelNumber-XYZ' in response to 'Acc model number'
27		GetAccessoryInfo	requesting 'Acc serial number'
28	RetAccessoryInfo		returning 'SerialNumber-1234' in response to 'Acc serial number'
29		GetAccessoryInfo	requesting 'Acc incoming max payload size'
30	RetAccessoryInfo		returning '1024 bytes' in response to 'Acc incoming max payload size'

Table C-54 Non-IDPS radio tagging command sequence

Step	Accessory command	Apple device command	Comment
Steps A–C cancel any current authentication process; see "Canceling a Current Authentication Process With IdentifyDeviceLingoes" (page 135). Step B also detects if the attached Apple device is a 3G iPod.			
A	`IdentifyDeviceLingoes`		identifying lingo 'General'; options none; device ID 0x00000000
The accessory waits up to 1 second for the Apple device to respond.			
B		`ACK`	acknowledging 'Success (OK)' to command 'General Lingo::IdentifyDeviceLingoes'
If the Apple device does not send an `ACK` command within 1 second, it may be a 3G iPod. In this case, if the accessory needs to support the 3G iPod and it is using UART communication, in may retry Step A up to 3 times, for a total elapsed time of 3 seconds. If the accessory does not receive an `ACK` command after the third try, it may either quit or send an `Identify` command to achieve limited use of iAP. See "The 3G iPod" (page 527) and "Interfacing With the 3G iPod" in *MFi Accessory Hardware Specification*.			
C		`GetAccessoryInfo`	the accessory ignores this query
D	`RequestTransportMaxPayloadSize`		requesting transport maximum payload size
E		`ReturnTransportMaxPayloadSize`	returning transport maximum payload size
The accessory now proceeds to identify its lingoes to the attached Apple device.			
1	`IdentifyDeviceLingoes`		identifying lingoes 'General'; options 'auth:none; power:low'; device ID 0x00000000
2		`ACK`	acknowledging 'Success (OK)' to command 'General Lingo::IdentifyDeviceLingoes'
3	`GetiPodOptions`		no params
4		`RetiPodOptions`	returning 'video output|line out'
5	`RequestLingoProtocolVersion`		requesting General Lingo version
6		`ReturnLingoProtocolVersion`	returning General Lingo v1.09
7	`RequestLingoProtocolVersion`		requesting Storage Lingo version
8		`ReturnLingoProtocolVersion`	returning Storage Lingo v1.02

Step	Accessory command	Apple device command	Comment
9	IdentifyDevice-Lingoes		identifying lingoes 'General|Storage'; options 'auth:immediate; power:low'; device ID 0x00000200
10		ACK	acknowledging 'Success (OK)' to command 'General Lingo::IdentifyDeviceLingoes'
11		GetAccessory-AuthenticationInfo	no params
12	RetAccessory-AuthenticationInfo		returning auth protocol v2.0; section: 0/1; cert data: ...
13		ACK	acknowledging 'Success (OK)' to command 'General Lingo::RetAccessoryAuthenticationInfo'
14	RetAccessory-AuthenticationInfo		returning auth protocol v2.0; section: 1/1; cert data: ...
15		AckDevAuthentication-Info	acknowledging 'auth info supported'
16		GetAccessoryInfo	requesting 'Acc info capabilities'
17		GetAccessory-Authentication-Signature	offering challenge '...' with retry counter 1
18	RetAccessory-Authentication-Signature		returning signature '...'
19		AckDevAuthentication-Status	acknowledging authentication status 'Success (OK)'
20	RetAccessoryInfo		returning 'Acc info capabilities|Acc name|Acc FW version|Acc HW version|Acc manufacturer|Acc model number|Acc incoming max payload size' in response to 'Acc info capabilities'
21		GetAccessoryInfo	requesting 'Acc name'
22	RetAccessoryInfo		returning 'Radio' in response to 'Acc name'
23		GetAccessoryInfo	requesting 'Acc FW version'
24	RetAccessoryInfo		returning 'v1.0.0' in response to 'Acc FW version'

Step	Accessory command	Apple device command	Comment
25		GetAccessoryInfo	requesting 'Acc HW version'
26	RetAccessoryInfo		returning 'v1.0.0' in response to 'Acc HW version'
27		GetAccessoryInfo	requesting 'Acc manufacturer'
28	RetAccessoryInfo		returning 'Radio Manufacturer' in response to 'Acc manufacturer'
29		GetAccessoryInfo	requesting 'Acc model number'
30	RetAccessoryInfo		returning 'M78901LL/Z' in response to 'Acc model number'
31		GetAccessoryInfo	requesting 'Acc incoming max payload size'
32	RetAccessoryInfo		returning '2048 bytes' in response to 'Acc incoming max payload size'
33	GetiPodCaps		no params
34		RetiPodCaps	returning 'total space 2147483648; max file size 1073741824; max write size 500; v1.02'
35	GetiPodFreeSpace		no params
36		RetiPodFreeSpace	returning 130023424 bytes
37	OpeniPodFeatureFile		opening feature type 'Radio Tagging'
38		RetiPodFileHandle	returning file handle 0
39		iPodAck	acknowledging 'Success' to command 'Storage Lingo::WriteiPodFileData' with file handle 0
40	CloseiPodFile		closing file with handle 0
41		iPodAck	acknowledging 'Success' to command 'Storage Lingo::CloseiPodFile' with file handle 0

Table C-55 Non-IDPS cardio equipment command sequence

Step	Accessory command	Apple device command	Comment
Steps A–C cancel any current authentication process; see "Canceling a Current Authentication Process With IdentifyDeviceLingoes" (page 135). Step B also detects if the attached Apple device is a 3G iPod.			

Step	Accessory command	Apple device command	Comment
A	IdentifyDeviceLingoes		identifying lingo 'General'; options none; device ID 0x00000000
The accessory waits up to 1 second for the Apple device to respond.			
B		iPodAck	acknowledging 'Success (OK)' to command 'General Lingo::IdentifyDeviceLingoes'
If the Apple device does not send an iPodAck command within 1 second, it may be a 3G iPod. In this case, if the accessory needs to support the 3G iPod and it is using UART communication, in may retry Step A up to 3 times, for a total elapsed time of 3 seconds. If the accessory does not receive an iPodAck command after the third try, it may either quit or send an Identify command to achieve limited use of iAP. See "The 3G iPod" (page 527) and "Interfacing With the 3G iPod" in *MFi Accessory Hardware Specification*.			
C		GetAccessoryInfo	the accessory ignores this query
D	RequestTransportMaxPayloadSize		requesting transport maximum payload size
E		ReturnTransportMaxPayloadSize	returning transport maximum payload size
The accessory now proceeds to identify its lingoes to the attached Apple device.			
1	IdentifyDeviceLingoes		identifying lingoes 'General'; options 'auth:none; power:low'; device ID 0x00000000
2		iPodAck	acknowledging 'Success (OK)' to command 'General Lingo::IdentifyDeviceLingoes'
3	GetiPodOptions		no params
4		RetiPodOptions	returning 'video output|line out'
5	RequestLingoProtocolVersion		requesting General Lingo version
6		ReturnLingoProtocolVersion	returning General Lingo v1.09
7	RequestLingoProtocolVersion		requesting Sports Lingo version
8		ReturnLingoProtocolVersion	returning Sports Lingo v1.01
9	RequestLingoProtocolVersion		requesting Storage Lingo version

Step	Accessory command	Apple device command	Comment
10		ReturnLingoProtocol-Version	returning Storage Lingo v1.02
11	IdentifyDevice-Lingoes		identifying lingoes 'General|Sports|Storage'; options 'auth:immediate; power:low'; device ID 0x00000200
12		iPodAck	acknowledging 'Success (OK)' to command 'General Lingo:: IdentifyDeviceLingoes'
13		GetAccessory-AuthenticationInfo	no params
14	RetAccessory-AuthenticationInfo		returning auth protocol v2.0; section: 0/1; cert data: ...
15		iPodAck	acknowledging 'Success (OK)' to command 'General Lingo:: RetAccessoryAuthenticationInfo'
16	RetAccessory-AuthenticationInfo		returning auth protocol v2.0; section: 1/1; cert data: ...
17		AckDevAuthentication-Info	acknowledging 'auth info supported'
18		GetAccessoryiceCaps	no params
19		GetAccessoryInfo	requesting 'Acc info capabilities'
20		GetAccessory-Authentication-Signature	offering challenge '...' with retry counter 1
21	RetAccessory-Authentication-Signature		returning signature '...'
22		AckDevAuthentication-Status	acknowledging authentication status 'Success (OK)'
23		GetAccessoryiceCaps	no params
24	RetAccessoryiceCaps		returning 'Gym equipment command support' with max 0 node filters
25		GetAccessoryiceCaps	no params

Step	Accessory command	Apple device command	Comment
26	RetAccessoryiceCaps		returning 'total space 0; max file size 0; max write size 0; read-only; no subdirs; random writes allowed; max file count 0; max name length 0; file system type Reserved; v1.02'
27	RetAccessoryInfo		returning 'Acc info capabilities| Acc name| Acc FW version| Acc HW version| Acc manufacturer| Acc model number| Acc incoming max payload size' in response to 'Acc info capabilities'
28		GetAccessoryInfo	requesting 'Acc name'
29	RetAccessoryInfo		returning 'Bike' in response to 'Acc name'
30		GetAccessoryInfo	requesting 'Acc FW version'
31	RetAccessoryInfo		returning 'v1.0.0' in response to 'Acc FW version'
32		GetAccessoryInfo	requesting 'Acc HW version'
33	RetAccessoryInfo		returning 'v1.0.0' in response to 'Acc HW version'
34		GetAccessoryInfo	requesting 'Acc manufacturer'
35	RetAccessoryInfo		returning 'Bike Manufacturer' in response to 'Acc manufacturer'
36		GetAccessoryInfo	requesting 'Acc model number'
37	RetAccessoryInfo		returning 'M78901LL/Z' in response to 'Acc model number'
38		GetAccessoryInfo	requesting 'Acc incoming max payload size'
39	RetAccessoryInfo		returning '2048 bytes' in response to 'Acc incoming max payload size'
40	GetiPodCaps		no params
41		RetiPodCaps	returning 'Gym equipment support| User data support' with userCount of 1
42	GetiPodCaps		no params

Step	Accessory command	Apple device command	Comment
43		RetiPodCaps	returning 'total space 2147483648; max file size 1073741824; max write size 500; read-only; no subdirs; random writes allowed; max file count 10000; max name length 200; file system type HFS+; v1.02'
44	GetUserIndex		no params
45		RetUserIndex	returning userIndex of 0
46	GetUserData		requesting 'Preferred unit system'
47		RetUserData	returning 'No information' for data type 'Preferred unit system'
48	GetUserData		requesting 'Name'
49		RetUserData	returning 'NewUser' for data type 'Name'
50	GetUserData		requesting 'Gender'
51		RetUserData	returning 'No information' for data type 'Gender'
52	GetUserData		requesting 'Weight'
53		RetUserData	returning '92.0 kg' for data type 'Weight'
54	GetUserData		requesting 'Age'
55		RetUserData	returning '26 years' for data type 'Age'
56	GetUserData		requesting 'Recording preference'
57		RetUserData	returning 'No information' for data type 'Recording preference'
58	OpeniPodFeatureFile		opening feature type 'Gym Equipment Workout' with options mask 'file data| ipodInfo| XML signature' and file data:</gymData>;
59		RetiPodFileHandle	returning file handle 0

Step	Accessory command	Apple device command	Comment
60	WriteiPodFileData		writing 293 bytes at offset 0 and handle 0: <?xml version=""1.0"" encoding=""UTF-8""?>"; <gymData>; <vers>1</vers>; <equipmentInfo>; <manufacturerID>00000001 </manufacturerID>; <manufacturerName>Bike Manufacturer </manufacturerName>; <type>Bike</type>; <model>M78901LL/Z </model>; <serialNumber> UV1234567890-SN </serialNumber>; </equipmentInfo>;
61		iPodAck	acknowledging 'Success' to command 'Storage Lingo:: WriteiPodFileData' with file handle 0
62	WriteiPodFileData		writing 125 bytes at offset 240 and handle 0: <userInfo><kg> 92.0</kg></userInfo>; <template>; <templateName>Athletic Challenge</templateName>; <sec>60</sec>; </template>;
63		iPodAck	acknowledging 'Success' to command 'Storage Lingo:: WriteiPodFileData' with file handle 0
64	WriteiPodFileData		writing 125 bytes at offset 366 and handle 0: <interval>; <event>start</event>; <sec>0</sec>; <kCal>0</kCal>; <km>0.00</km>; <rpm>26</rpm>; <level>1</level>; </interval>;
65		iPodAck	acknowledging 'Success' to command 'Storage Lingo:: WriteiPodFileData' with file handle 0
66	WriteiPodFileData		writing 86 bytes at offset 392 and handle 0: <interval>; <sec>10</sec>; <kCal>0</kCal>; <km>0.02</km>; <rpm>64</rpm>; </interval>;
67		iPodAck	acknowledging 'Success' to command 'Storage Lingo:: WriteiPodFileData' with file handle 0
68	WriteiPodFileData		writing 87 bytes at offset 479 and handle 0: <interval>; <sec>20</sec>; <kCal>12</kCal>; <km>0.7</km>; <rpm>189</rpm>; </interval>;

Step	Accessory command	Apple device command	Comment
69		iPodAck	acknowledging 'Success' to command 'Storage Lingo::WriteiPodFileData' with file handle 0
70	WriteiPodFileData		writing 107 bytes at offset 567 and handle 0: <interval>; <event>end</event>; <sec>21</sec>; <kCal>13</kCal>; <km>0.8</km>; <rpm>170</rpm>; </interval>;
71		iPodAck	acknowledging 'Success' to command 'Storage Lingo:: WriteiPodFileData' with file handle 0
72	WriteiPodFileData		writing 99 bytes at offset 675 and handle 0: <workoutSummary>; <sec>21</sec>; <kCal>13</kCal>; <km>0.8</km>; <rpm>170</rpm>; </workoutSummary>;
73		iPodAck	acknowledging 'Success' to command 'Storage Lingo:: WriteiPodFileData' with file handle 0
74	CloseiPodFile		closing file with handle 0
75		iPodAck	acknowledging 'Success' to command 'Storage Lingo:: CloseiPodFile' with file handle 0

Glossary

accessory A third-party device licensed under the Made for iPod program.

authentication A mechanism used by an Apple device to verify whether an attached accessory is an authorized accessory and by an accessory to authenticate the Apple device, if desired.

checksum The byte sum of packet bytes from the payload length through the last packet byte. This is used to validate the contents of a command packet. For a valid packet, the sum of the bytes, including the checksum byte, must be `0x00`. The packet checksum byte—the last byte in a packet—must be the 2's complement (the negative) of the sum of the payload length byte up to, but not including, the packet checksum byte.

deprecated Used to describe a technology or feature that is supported but whose use is discouraged and not recommended. Such a technology or feature has typically been replaced by a newer one and is likely to become unsupported in the future.

FID A Full ID string, used by IDPS to send token-value fields that an Apple device is able to parse during the accessory identification process.

HID (Human Interface Device) HID is a standard USB class. A USB host such as a PC or Macintosh will recognize any attached USB device that supports a HID interface and makes it available to the application layers of the operating system via a set of programming interfaces. A common application of a HID interface is a USB mouse or joystick.

HID report A single unit of data that is used to send information to the HID interface of an Apple device or from the Apple device to the host. iAP packets are broken into HID reports before being sent across the USB port link and are reassembled on the receiving side.

IDPS Identify Device Preferences and Settings, the identification process required for an accessory to communicate with an Apple device. See "Identification" (page 93).

iUI (iPod USB Interface) A configuration of an Apple device when attached as a device over USB. This configuration allows the Apple device to be controlled using iAP, using a USB HID class interface as a transport mechanism.

LCB (Link Control Byte) A byte used by the iUI to indicate report sets and manage data flow.

lingo The command category used by an accessory. There is a General lingo that must be supported by all accessories. Other lingoes are designed for use by specific accessories, such as simple remote controls and microphones.

link The logical connection between an external accessory and an Apple device via serial port or other physical connection.

low power mode An operating mode of an accessory in which it draws low power from an attached Apple device, as defined in the section "Accessory Power Policy" in *MFi Accessory Hardware Specification*.

packet The logical set of bytes that compose a valid command sequence. This set includes the packet start byte, packet payload length, payload, and payload checksum. Note that a sync byte is appended to the beginning of the packet when using the UART serial port as the data transport. There are two different packet types: small format and large format.

payload The sequence of bytes consisting of the lingo, command, and data that are contained within a packet.

podcasting A way to publish multimedia files on the Internet that lets users receive new files automatically by subscription. Podcast files are typically downloaded to Apple devices through Apple's iTunes application.

RDS/RBDS (Radio [Broadcast] Display System) A technology for broadcasting and displaying artist, album, track titles, and similar information on FM radio receivers.

resistor-based accessory An accessory that uses an Accessory Identify resistor to access only limited functions in an Apple device. Compare Serial accessory.

RSSI (Receive Signal Strength Indicator) A measure of the strength of an RF signal coming into a radio frequency tuner.

serial accessory An accessory that uses the Accessory Protocol Interface to access a range of Apple device functions. Compare Resistor-based accessory.

UART (Universal Asynchronous Receiver/Transmitter) A piece of computer hardware that translates between parallel and serial bits of data. A UART is usually an integrated circuit used for serial communications over a computer or peripheral accessory serial port.

USB (Universal Serial Bus) An interface standard for communication between a computer and external peripherals over a cable using biserial transmission.

USB descriptor A standard USB data structure that is passed from a USB device to the host upon request. Descriptors are used by the USB device to communicate its characteristics and resource requirements to the host.

USB endpoint A logical connection point that is used to set up a data transfer pipe between a USB host and interface on an accessory. For instance, the HID interface on Apple devices uses an interrupt-type endpoint to enable a pipe for transferring data to the USB Host.

USB host A single computer connected to one or more USB devices or functions. The host is responsible for recognizing that a USB device has been attached to it and for driving the communications with the device. For the purposes of this document, an Apple device is a USB device that provides a function, and an accessory is the USB host. However, an Apple device can also be made a USB host with the accessory acting as its USB device.

X.509 certificate A standard defined by the International Telecommunication Union (ITU) that governs the format of certificates used for authentication and sender identity verification in public-key cryptography. In the iAP, X.509 certificates contain the public keys used in the authentication process.

Document Revision History

This table describes the changes to *MFi Accessory Firmware Specification*.

Date	Notes
2012-09-12	*Revision R46:* Updated for iOS 6.0.
	Added a Resume iPod command code to Table 5-49 (page 429).
	Deprecated "Command 0x0032: SetDisplayImage" (page 435).
	Deprecated the Nike + iPod Cardio technology.
	Numbered all sections.
2012-02-21	*Revision R45:* Updated for iPad (3rd generation).
	Removed references in "Command 0x0C: RotationInputStatus" (page 233) to jog wheels, which are not supported.
	Added four-byte command requirement to "Command 0x0026: SetPlayStatusChangeNotification" (page 424).
	Revised description of "Command 0x54: SetAvailableCurrent" (page 204) to let Apple devices draw higher current limits.
	Clarified the one-byte parameter value passed by "Command 0x64: RequestApplicationLaunch" (page 205).
	Removed references to nonexistent iPod Out lingo command `iPodOutScreenChange`.
2011-10-14	*Revision R44:* Restructured document; updated it for iOS 5.0 and iPhone 4S.
	Changed the formats of all packet tables in the iAP command descriptions to cite generic header and footer specifications given in a new section, "Command Packets" (page 109).
	Renamed `RequestRemoteUIMode` to `RequestExtendedInterfaceMode`, `ReturnRemoteUIMode` to `ReturnExtendedInterfaceMode`, `EnterRemoteUIMode` to `EnterExtendedInterfaceMode`, and `ExitRemoteUIMode` to `ExitExtendedInterfaceMode`.
	Renamed `RetFIDTokenValueACKs` to `AckFIDTokenValues`.
	Renamed `SendHIDReportToiPod` to `iPodHIDReport` and `SendHIDReportToAcc` to `AccessoryHIDReport`.

Date	Notes
	Renamed `DevDataTransfer` to `AccessoryDataTransfer`.
	Renamed `GetDevAuthenticationInfo` to `GetAccessoryAuthenticationInfo`, `RetDevAuthenticationInfo` to `RetAccessoryAuthenticationInfo`, `AckDevAuthenticationInfo` to `AckAccessoryAuthenticationInfo`, `GetDevAuthenticationSignature` to `GetAccessoryAuthenticationSignature`, `RetDevAuthenticationSignature` to `RetAccessoryAuthenticationSignature`, `AckDevAuthenticationStatus` to `AckAccessoryAuthenticationStatus`, `GetAccSampleRateCaps` to `GetAccessorySampleRateCaps`, `RetAccSampleRateCaps` to `RetAccessorySampleRateCaps`, `SetAccStatusNotification` to `SetAccessoryStatusNotification`, `RetAccStatusNotification` to `RetAccessoryStatusNotification`, `AccCapsToken` to `AccessoryCapsToken`, and `AccInfoToken` to `AccessoryInfoToken`.
	Renamed `GetDeviceVersion` to `GetAccessoryVersion`, `RetDeviceVersion` to `RetAccessoryVersion`, `GetDeviceCaps` to `GetAccessoryCaps`, `RetDeviceCaps` to `RetAccessoryCaps`, `DevStateChangeEvent` to `AccessoryStateChangeEventCaps`, `GetDevCaps` to `GetAccessoryCaps`, `RetDevCaps` to `RetAccessoryCaps`, `GetDevControl` to `GetAccessoryControl`, `RetDevControl` to `RetAccessoryControl`, `SetDevControl` to `SetAccessoryControl`, `GetDevData` to `GetAccessoryData`, `RetDevData` to `RetAccessoryData`, `SetDevData` to `SetAccessoryData`, and `AsyncDevData` to `AsyncAccessoryData`.
	Renamed `AccessibilityEvent` to `VoiceOverEvent`, `GetAccessibilityParameter` to `GetVoiceOverParameter`, `RetAccessibilityParameter` to `RetVoiceOverParameter`, `SetAccessibilityParameter` to `SetVoiceOverParameter`, `GetCurrentItemProperty` to `GetCurrentVoiceOverItemProperty`, `RetCurrentItemProperty` to `RetCurrentVoiceOverItemProperty`, `SetContext` to `SetVoiceOverContext`, and `AccessibilityParameterChanged` to `VoiceOverParameterChanged`.
	Renamed notifications `NowPlayingFocusApp` to `NowPlayingApplicationBundleName` and `NowPlayingFocusAppName` to `NowPlayingApplicationDisplayName`.
	Renamed commands `GetNowPlayingFocusApp` and `RetNowPlayingFocusApp` to `GetNowPlayingApplicationBundleName` and `RetNowPlayingApplicationBundleName`.
	Conformed the names of all acknowledgment commands sent by an accessory to be `AccessoryAck`, and all acknowledgment commands sent by an Apple device to be `iPodAck`.
	Changed references to “Extended Interface protocol” to “Extended Interface mode.”

Date	Notes
	Split contents of former chapter "The Protocol Core and the General Lingo" into two new chapters: "Protocol Core" (page 87) and "The General Lingo" (page 119).
	Added new section "Extended Interface Mode" (page 56) to Chapter 2, "Protocol Features and Availability," and incorporated explanatory text from Chapter 5.
	Added new section "Starting IDPS" (page 96).
	Added new section "Determining Apple Device Capabilities" (page 97).
	Added section "Status Notifications From Accessories" (page 49) to Chapter 2, "Protocol Features and Availability."
	Added section "iTunes Tagging" (page 83) to Chapter 2, "Protocol Features and Availability" and changed name of Appendix A to "iTunes Tagging Accessory Design."
	Added section "Nike + iPod Cardio Equipment System" (page 85) to Chapter 2, "Protocol Features and Availability" and changed name of Appendix B to "Nike + iPod Cardio Accessory Design."
	Added new section "Apple Device Language" (page 117) and associated General lingo commands.
	Added new section "Wi-Fi Network Login Sharing" (page 56) and associated General lingo commands.
	Enlarged and retitled section "Bluetooth Autopairing and Connection Status Notifications" (page 55), and added new Bluetooth connection status notifications; see Table 3-126 (page 190).
	Added section "Status Notifications From Apple Devices" (page 48) to replace section "Apple Device Event Notifications."
	Expanded section "User Interface Accessibility" and renamed it "VoiceOver" (page 81).
	Added sensor information to Apple device model descriptions in Table I-1 (page 32).
	Added new paragraph to "iPod Out Mode" (page 78) advising accessories to register for iPod Out Mode status change notifications.
	Integrated General lingo commands `RequestTransportMaxPayloadSize` and `ReturnTransportMaxPayloadSize` into all accessory identification processes; see "Sample IDPS Command Sequences" (page 98) and "Accessory Identification With Non-IDPS Apple Devices" (page 553).
	Added new General lingo "Command 0x56: SetInternalBatteryChargingState" (page 204).

Date	Notes
	Added new Extended Interface "Command 0x004A: PrepareUIDList" (page 457) through "Command 0x004F: RetArtworkData" (page 460). Four of these commands supersede existing Display Remote lingo commands `GetTrackArtworkTimes`, `RetTrackArtworkTimes`, `GetTrackArtworkData`, and `RetTrackArtworkData`.
	Deprecated "Command 0x05: EnterExtendedInterfaceMode" (page 532). New accessory designs should use General lingo "Command 0x0017: SelectDBRecord" (page 415) instead.
	Deprecated "Command 0x0028: PlayCurrentSelection" (page 545). New accessory designs should use Extended Interface lingo "Command 0x0017: SelectDBRecord" (page 415) instead.
	Deprecated iPod Out lingo "Command 0x0028: PlayCurrentSelection" (page 545) as no longer useful.
	Removed power control bit documentation from Table 3-25 (page 135) and Table 3-69 (page 163). The functions of these bits are superseded by the Accessory Power Policy in *MFi Accessory Hardware Specification*.
	Removed bit 12, "Database Available," from Table 3-112 (page 184); it is no longer supported.
	Moved section "Using an Apple Device as a USB Host" to new "Protocol Core" chapter and renamed it "USB Host Mode" (page 88).
	Moved content of former appendix "Accessory Identification" to new "Protocol Core" chapter, section "Identification" (page 93).
	Moved content of former appendix "Transaction IDs" to new "Protocol Core" chapter, section "Transaction IDs" (page 110).
	Moved content of former appendix "Multisection Data Transfers" to new "Protocol Core" chapter, section "Multisection Data Transfers" (page 116).
	Moved Table 1-17 (page 81) (list of Simple Remote lingo VoiceOver commands) from Chapter 4 to Chapter 2.
	Changed name of former chapter "Accessory Lingoes" to "Additional Lingoes" (page 211).
	Changed name of former section "Accessory Signaling and Initialization" to "Transport Initialization" (page 88).
	Changed name of former section "Packet Signaling and Initialization Using the UART Serial Port Link" to "UART" (page 88).
	Changed name of former section "iAP Signaling and Initialization Using the USB or BT Port Link" to "USB Device Mode" (page 93).
	Changed terminology to refer to "transports" instead of "transport links."

Date	Notes
	Clarified the required accessory response to `RetiPodAuthenticationInfo`.
2011-04-04	*Revision R43:* Updated for iPad 2, iPhone 4 (CDMA model), and iOS 4.3.1.
	Updated "Notice of Proprietary Property" (page 31).
	Updated General lingo `iPodAck` command formats.
	Added General lingo bit 30 to `RetiPodOptionsForLingo` values; see Table 3-132 (page 192).
	Deprecated and removed General lingo commands 0x0D, `RequestiPodModelNum`, and 0x0E, `ReturniPodModelNum`.
	Changed names of General lingo commands `RequestTransportMaxPacketSize` and `ReturnTransportMaxPacketSize` to `RequestTransportMaxPayloadSize` and `ReturnTransportMaxPayloadSize`.
	Incorporated numerous other updates and corrections.
2010-11-29	*Revision R42:* Updated for iOS 4.2.1.
	Corrected generic references to Apple devices, iOS devices, and iPods throughout the document; see "IMPORTANT" (page 31).
	Globally replaced `SDKProtocolToken` and `SDKProtocolMetadataToken` with `EAProtocolToken` and `EAProtocolMetadataToken`.
	Added new section "iPod Out Mode" (page 78).
	Added new section "Upgrading from UART Transport to USB Host Mode" (page 89).
	Updated specifications of `PlayCurrentSelection`.
	Restored documentation of USB Host mode power management command "Command 0x54: SetAvailableCurrent" (page 204).
	Restored documentation of USB Host mode charging notifications; see Table 3-118 (page 188).
	Reserved Accessory Info Type 0x02; see Table 3-42 (page 145) and Table 3-46 (page 147).
	Deprecated the Microphone lingo; see "Deprecated Lingo 0x01: Microphone Lingo" (page 532).
	Deprecated "Command 0x0038: SelectSortDBRecord" (page 546).
	Deprecated chapter play control commands listed in Table 5-49 (page 429).

Date	Notes
	Removed section "Accessory Power Policy" from Chapter 2 and inserted a reference to the same section of *MFi Accessory Hardware Specification*.
2010-09-08	*Revision R41:* Updated for 6G nano, 4G touch, and iOS 4.1.
	Added Display Remote lingo commands 0x21 and 0x22 to support Genius playlists; see Table 4-48 (page 249).
	Added Extended Interface lingo commands 0x44-0x45 and 0x47-0x49 to support Genius playlists; see Table 5-1 (page 398).
	Added new section "Playback Status Notifications" (page 59).
	Added new sections "Audio and Video Output Preferences" (page 47) and "Audio/Video Synchronization."
	Documented new USB HID commands (see Table 4-12 (page 225)).
	Deprecated iPod alarm events; see Table 4-59 (page 255).
	Changed name of Digital Audio lingo "Command 0x04: TrackNewAudioAttributes" (page 356) from `NewiPodTrackInfo` to `TrackNewAudioAttributes`.
	Noted that `SelectSortDBRecord` is not supported on iPods that run iOS.
	Added transaction ID fields to `GetDevAuthenticationSignature` and `RetDevAuthenticationSignature` packets.
	Corrected payload size range of large iAP packets; see "Large Packet Format."
	Revised General lingo command 0x0E, `ReturniPodModelNum`, to show that it no longer returns meaningful model IDs.
	Clarified generic references to iPods and iOS devices; see "IMPORTANT" (page 31).
2010-06-24	*Revision R40:* Updated for iPhone 4.
	Changed document name from "iPod Accessory Protocol Interface Specification" to "MFi Accessory Firmware Specification."
	Changed name of "iPhone OS" to "iOS" for version 4 and later versions.
	Added information for 3rd generation iPod touch.
	Added section "Lingo 0x0D: iPod Out Lingo" (page 365).
	Added section "Accessory Launching of iOS Applications" (page 50).
	Added section "USB Human Interface Device Reports" (page 225).
	Added section "User Interface VoiceOver Commands."

Date	Notes
	Added section "Interaction with iOS Media Applications" (page 54).
	Updated section "USB Host Mode" (page 88).
	Added General lingo "Command 0x11: RequestTransportMaxPayloadSize" (page 132) and "Command 0x12: ReturnTransportMaxPayloadSize" (page 132).
	Added General lingo "Command 0x50: CancelCommand" (page 203) for multipacket data transfers.
	Added new notifications; see "Command 0x4A: iPodNotification" (page 185).
	Documented `SessionWriteFailure` error for large data transfers; see Table 3-9 (page 127).
	Documented accessory RF certification declarations; see Table 3-74 (page 166).
	Documented application matching controls; see Table 3-81 (page 168).
	Documented `ScreenInfoToken` for iPod Out mode.
	Deprecated Simple Remote lingo "Command 0x02: ImageButtonStatus" (page 542).
	Incorporated numerous other updates and corrections.
2010-04-09	*Revision R39:* Updated for iPad.
	Added new appendix A, "iAP Interfaces for the iPad".
	Added USB Host mode power management commands 0x52–0x54, 0x56, and 0x58–0x59 to the General lingo, as listed in Table 3-1 (page 119).
	Added commands 0x46–0x48 to the General lingo to support accessory status notifications, as listed in Table 3-1 (page 119), and added Status Info type 0x0B to `GetAccessoryInfo` and `RetAccessoryInfo`.
	Restructured Appendix E, "iTunes Tagging Accessory Design" (page 467), and updated it for additional broadcast sources.
	Incorporated numerous updates and corrections.
2009-10-22	*Revision R38:*
	Added section "Reserved Commands and Data" (page 87).
	Added information for the 5G nano, the 2G touch (2009), and the iPod classic 160 GB.
	Added commands 0x49-0x4A, 0x4D-0x4F, and 0x51 to the General lingo to support Event Notifications; see Table 3-1 (page 119). Also added explanatory section "User Interface Restrictions" (page 528).

Date	Notes
	Added commands 0x0D and 0x0E to the Simple Remote lingo to support button actions for 5G nano Radio Tagging and Camera accessories; see Table 4-3 (page 213).
	Added section "Accessory Control of the iPod 5G nano Camera" (page 218).
	Added commands 0x25-0x31 to the RF Tuner lingo, and added features to other commands, to support HD radio; see Table 4-111 (page 288).
	Incorporated numerous updates and corrections.
2009-09-09	*Revision R37:*
	Exported the following chapters and appendixes to new book *iPod/iPhone Hardware Specifications*, Release R1: "Hardware Interfaces," "Functional Description," "Protocol Transports," "iPod Power States and Accessory Power," "Headphone Remote and Mic System," "Interfacing With the 3G iPod," "Sample Accessory Circuits," "Power Guidelines," and "FireWire to USB Reference Design."
	Exported appendix "Headset and FireWire-to-USB Power Converter Certification" to new book *iPod/iPhone Accessory Testing and Certification Specification*, Release R1.
	Imported entire contents of discontinued book *iPod Extended Interface Specification*, Release R25, into "Extended Interface Mode" (page 397).
	Imported some content from discontinued book *iPhone Accessory Interface Specification*, Release R9.
	Reformatted content to new layout and typography.
2009-06-23	*Revision R36:*
	Added information about the iPhone 3GS.
	Added new appendix "Accessory Identification." *This identification process is required in all new accessory designs.* Moved command examples using `IdentifyDeviceLingoes` to "Accessory Identification With Non-IDPS Apple Devices" (page 553) in Appendix M, "Historical Information."
	In Chapter 3, added new section "Accessory Communication With iOS Applications" (page 50); deleted section "iPod Games" and updated section "Accessory Control of iOS Devices" (page 49).
	Added commands 0x38-0x3C, 0x3F-0x43, and 0x4A-0x4C to the General lingo; see Table 3-1 (page 119).
	Added new section "Accessory Power Policy."
	Added new section "Lingo 0x0E: Location Lingo" (page 369).
	Added sample command sequences for iTunes tagging (Table A-6 (page 476)) and cardio equipment (Table B-7 (page 516)).

Date	Notes
	Added new appendix "Transaction IDs" (page 110).
	Added new appendix "Multisection Data Transfers" (page 116).
	Added new section "Avoiding an iPhone OS Warning When a Self-Powered Accessory Is Off."
	Added caution about latching connectors to "30-Pin Connector."
	Changed button press detection parameters in "iPod/iPhone Headphone/Microphone Jack."
	Added information about PKCS-7 transport of certificate data to "Accessory Authentication of the Apple Device" (page 108).
	Added warning about the iPod touch and iPhone Off state to "Power States."
	Updated signal level information in "UART Serial Port Link."
	Added new section "Examples of Transaction IDs" (page 112).
	Added new section "Sample Identification Sequences" (page 389).
2009-03-17	*Revision R35:*
	Added Table I-1 (page 32) to identify iPod/iPhone models.
	Revised "iPhone headphone/microphone schematic" to show iPod/iPhone differences.
	Revised section "Line Level Input" in Chapter 2.
	Updated Table 1-3 (page 42) to show current firmware versions.
	Added display resolution data to Table 1-8 (page 47).
	Updated and revised accessory identification and authentication in Table C-47 (page 552).
	Added new `iPodAck` error codes to Table 3-6 (page 125).
	Added copy protection requirement to "USB Device Mode Audio" (page 347).
	Revised section "Button Detection Circuitry" in Appendix C.
2009-01-05	*Revision R34:*
	Added definitions of legal agreement terminology in Introduction.
	Added new appendix "Accessory Certification."
	Added FM radio tagging information to "iTunes Tagging Accessory Design" (page 467).

Date	Notes
	Added new section "Lingo 0x09: Sports Lingo" (page 335).
	Added new appendix "Nike + iPod Cardio Accessory Design" (page 501).
	Added new appendix "Headphone Remote and Mic System."
	Added new commands 0x80-0x82 and new options for command 0x12 in "Lingo 0x0C: Storage Lingo" (page 357).
	Documented new models: the 4G iPod nano, 120 GB classic, and 2G iPod touch.
	Added section "Accessory Control of iOS Devices" (page 49).
	Added hardware details of the iPhone audio connection ("iPod/iPhone Headphone/Microphone Jack").
	Clarified USB power requirements in "USB 2.0."
	Added section "Magnetic Sensitivity of the iPod."
	Defined recommended procedure for determining iPod capabilities in "Command 0x13: IdentifyDeviceLingoes" (page 133).
	Added section "Playback Engine Playlists" (page 216).
	Clarified behavior of Next and Previous buttons in "Using Contextual Buttons" (page 216).
	Clarified mute event in Table 4-61 (page 257).
	Deprecated commands in "Deprecated USB Host Control Commands" (page 549).
	Deprecated "General Lingo Command 0x01: Identify" (page 530)).
	Removed from Chapter 5 obsolete description of testing for the General lingo over UART transport.
2008-06-26	*Revision R33:*
	Added section "Command Timings" (page 212).
	Revised audiobook playback speeds in Table 4-66 (page 263).
	Added accessory requirements to Table 3-48 (page 148).
	Changed names of levels of authentication from V1 and V2 to authentication 1.0 and 2.0.
	Changed name of RF Transmitter lingo (0x05) to Accessory Power lingo.
	Added section "Supplying USB Power" to Appendix A.

Date	Notes
	Changed "Data Transfer to the Apple Device" (page 471) in Appendix B to show all radio tagging plist fields required.
	Added section "Powering the 3G iPod" to Appendix C.
	Added Note about lyrics strings to Table 4-82 (page 273).
	Revised "Typical diode bridge circuit for an AC adapter" in Appendix E.
	Fixed typo in Table F-2 in Appendix F.
2008-05-07	*Revision R32:*
	Added new appendix, "FireWire to USB Reference Design."
	Revised documentation of sleep states in Chapter 2, Appendix A, and elsewhere.
	Added section "Video Output Settings" (page 47) and expanded Table 3-57 (page 152).
	Updated the current firmware versions in Table 1-2 (page 42).
	Made several updates to Appendix B, "iTunes Tagging."
	Made numerous other updates, corrections, and clarifications throughout the document.
2007-12-12	*Revision R31:*
	Moved past firmware version documentation, description of the 9-pin Audio/Remote connector, FireWire specifications, and other noncurrent material to new appendix, "Historical Information" (page 523).
	Documented November 2007 firmware for the 5G, classic, 3G nano, and touch iPod models.
	Made minor correction to "Configuration and interface descriptors for iPods with USB audio."
	Simplified document structure.
2007-10-02	*Revision R30:*
	Added "Lingo 0x0C: Storage Lingo" (page 357) to Chapter 6.
	Added new Appendix B, "iTunes Tagging Accessory Design" (page 467).
2007-09-05	*Revision R29:*
	Added documentation for the iPod classic, iPod 3G nano, and iPod touch.
	Added documentation for component video outputs.
	Added new command, `SetVideoDelay`, to the Digital Audio lingo.

Date	Notes
	Deprecated the FireWire interface on the 30-pin connector.
	Added new preference IDs to `GetiPodPreferences`.
	Added design guidelines for third-party developers of AC adapter accessories for the iPod touch (“Power Guidelines”).
	Added design guidelines for third-party developers of carrying cases for iPod touch (“iPod touch Carrying Case Design”).
2007-06-29	*Revision R28:*
	Revised pin connections in 30-pin to FireWire cable (“30-pin to FireWire cable”).
	Updated model listings to include iPhone and new iPod models.
	Updated requirements for artwork count data (Table 4-82 (page 273))
	Added caution about sending signals to the iPod UART when its serial receive block is off.
	Documented X.509 certificate classes (Table 2-6 (page 104))
	Added line-out usage controls (Table 3-57 (page 152))
	Added section “USB Device Mode Audio Errors on Older Apple Devices” (page 352)
	Added authentication requirement to General lingo commands 0x1A-0x1F
	Deprecated Level V1 authentication for new designs (see “Apple Device Authentication of the Accessory” (page 106))
	Deprecated “General Lingo Command 0x01: Identify” (page 530)
2007-02-06	*Revision R27:*
	Added section “Minimizing Crosstalk and Noise.”
	Added new information to Table 1-2 (page 42).
	Removed autobaud on parity errors from Table 1-5 (page 44).
	Clarified UART communication rates in “UART Serial Port Link.”
	Removed Manufacturer String and Product String from “Choosing an iPod USB Configuration.”
	Added example of using iAP over USB in “Transferring IdentifyDeviceLingoes and iPodAck commands over USB using iAP.”
	Distinguished behavior of audio and video playback when iPod enters Extended Interface mode using `EnterExtendedInterfaceMode`.

Date	Notes
	Added example of using Display Remote Protocol in "Lingo 0x03: Display Remote Lingo" (page 248).
	Clarified usage of `NewiPodTrackInfo` command in "Command 0x04: TrackNewAudioAttributes" (page 356).
	Added new appendix, "Interfacing With the 3G iPod."
	Added new appendix, "Sample Accessory Circuits."
	Added temperature range to "UART Serial Port Link."
2006-11-03	Added new information to Tables 3-1 and 5-26.
2006-10-17	*Revision R26:*
	Added new information to the note after Table 2-2.
	Updated Table 2-3.
	Added note to section "Line Level Output."
	Added new section "Headphone Jack on Video-Capable iPods."
	Added new iPod models and software versions.
2006-09-12	*Revision R25:*
	Added iPod options and preferences commands (0x24-0x25 and 0x29-0x2B) to General lingo.
	Added USB Host Control lingo (0x06).
	Added RF Tuner lingo (0x07).
	Updated list of model ID strings.
	Corrected `RetTrackArtworkData` packet listing (Table 4-89 (page 277)).
2006-06-19	*Revision R24:*
	Added Accessory Equalizer lingo, number 0x08.
	Added USB Digital Audio lingo, number 0x0A.
	Added Authentication level V2 (X.509 certification) to General Lingo (0x00).
	Added Album Art commands (0x16-0x19 and 0x1F-0x20) to Display Remote Lingo (0x03).
	Added GetAccessoryInfo (0x27) and RetAccessoryInfo (0x28) commands to the General lingo (0x00).

Date	Notes
	Added Dedicated Media commands (0x00-0x04) to the Simple Remote lingo (0x02).
	Added timeout and retry information to various command descriptions.
	Moved documentation of Extended Interface commands (0x03-0x06, General lingo) from the *iPod Extended Interface Specification*.
	Revised model and feature tables in Chapter 3.
	Added and updated lingo history tables in Chapter 6.
2006-02-10	*Revision R23:*
	Pages 17 and 27: Changed audio output power specification to 25 mW.
	Page 87: A simple remote accessory must send a data payload when all buttons are released; 200 ms timeout removed.
	Corrected Table 1-2 (page 42).
	Note, page 17: Specified Technical Note TN001i by name.
2006-01-05	*Revision R22:*
	General update and reorganization of content.
	Added information on iPod power states.
	Added "D+ and D- connections for a 500 mA USB power brick" and "iPod equivalent input circuits."
2005-10-12	Updated functional descriptions.
	Added new microphone lingo commands.
	Additional information about hibernate mode.
	Bug fixes for volume control and others.
2005-09-07	Added support for USB/iUI, authentication and additions to the new Display Remote lingo (0x03)
2005-03-21	Added Equalizer Control lingo support and commands.
2004-11-12	Added general lingo commands, bug fixes and pinouts for the iPod photo release
2004-08-03	Reserved the 28k pulldown resistor
2004-07-21	Added lingo command packet examples.
	Corrected doc properties.
	Added minor clarifications.

Date	Notes
	Added new accessory detect image.
	Updated content based on internal review.
2004-05-17	Incorporated review feedback
2004-04-20	Update simple remote, doc reformat
2004-01-14	Minor clarifications
2003-08-12	5mA access power note
2003-08-04	New serial, add bottom serial
2003-07-22	Picture to show Remote Data lines
2003-07-07	License agreement
2003-04-15	Remote protocols added
2003-04-02	Car Charger Detect added
2003-02-04	Accessory Detect Resistor Change
2003-01-09	Rx, Tx Clarification
2002-12-04	Initial Release.

Document Revision History