iOS ACCESSIBILITY HANDBOOK

iOS Accessibility Handbook

A clear, concise and complete reference.

Luis Abreu

This book is for sale at http://leanpub.com/iosaccessibility

This version was published on 2016-03-11

Tweet This Book!

Please help Luis Abreu by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought ”iOS Accessibility Handbook: A clear, concise and complete reference” and supported @watsi

The suggested hashtag for this book is #iosa11y.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#iosa11y

Contents

About the Author

Luis Abreu is a UX Designer, he believes cross-domain knowledge is key to building great products.

Interested in Design, Business, Psychology, and Development, he's got over 10 years of experience, working for small and large companies, doing web and native projects, client and server-side, the only constant is his drive for continuous improvement and doing what's best for consumers.

He publishes his work and articles on his personal website.

This is his first book.

Dedication

To all the people I've worked or interacted with throughout the years. To friends including the ones from my childhood, family, colleagues, teachers; everyone who made me see new things, even if I disagreed with them at first; everyone I still disagree with and drives me to find better solutions.

A special thanks to my girlfriend, Parisa.

Terms

Disability

Disability refers to a severe impairment causing the complete loss of a function, e.g. Blindness, Deafness, etc.

Impairment

Impairment refers to a partial loss of function, e.g. Color-Blindness, Hard-of-Hearing, Repetitive Strain Injury etc.

In addition to Severity, which can range from Mild to Moderate and Severe, Impairments also vary in Nature and Duration (Temporary/Permanent).

Impairment Nature (IN)

Impairment Nature refers to the origin of an Impairment.

Internal Impairment (II)

Internal Impairments originate from an individual's body or mind. Well-known permanent severe impairments such as Blindness, or Deafness, usually fall under this Impairment Nature.

External Impairment (EI)

External Impairments originate from an individual's current environment or activity. While they certainly can cause low-vision, low-hearing, or motor/cognitive limitations, the commonly temporary duration and mild severity, and origin in familiar environmental factors such as sunlight or noise, makes External Impairments easy to dismiss.

Hearing Impairment

Hearing Impairment refers to any condition limiting hearing, e.g. low-hearing, difficulty in discerning sounds such as speech.

Learning and Literacy Impairment

Learning and Literacy Impairment refers to any condition limiting the understanding of information, and ability to complete an action. Examples include not being able to focus on a task due to conditions such as autism or an activity such as driving, as well as not being able to understand information due to dyslexia or lack of familiarity with the terms used.

Physical and Motor Skills Impairment

Physical and Motor Skills Impairment refers to any limitation to physically interacting with a device. Either due to a complete inability to move; not being able to meet the precision and dexterity required to perform more advanced gestures; or require a longer period of time to do so. Examples include Repetitive Strain Injury which temporarily limits dexterity and speed, but also paraplegia requiring interaction to be done through a specialized button, potentially activated by mouth or eye.

Vision Impairment

Vision Impairment refers to any condition limiting vision, e.g. color-blindness, low-vision, blindness.

Accessibility Conformance Levels (ACL)

Accessibility Conformance Levels range from Single-A (essential), Double-A (good), and Triple-A (ideal).

Assistive Technologies (AT)

Technologies such as VoiceOver or Text Scaling that inform and help users interact with your app in the way that’s most accessible to them.

Accessibility Semantics (AS)

Accessibility Semantics expose and describe elements to Assistive Technologies so they can be read and interacted with.

Guided Access (GA)

Guided Access helps keep users focused on a single task by preventing navigating outside of the current app, disabling certain screen areas, device features such as volume, as well as app features. Guided Access is also known as Single App Mode.

Current status can be determined using `.IsGuidedAccessEnabled`, changes in status are broadcast as `.GuidedAccessStatusDidChangeNotification`, and apps can programmatically enable this feature, see `.RequestGuidedAccessSession` for details.

Switch Control (SC)

Switch Control changes VoiceOver granularity. People can jump between high-level Headings, individual Links/Lines/Forms, or go down to low-level items such as characters to understand spelling.

VoiceOver (VO)

VoiceOver refers to the Assistive Technology, included in all Apple products, which facilitates understanding of screen contents through means other than vision such as Text-to-Speech and Braille Displays, as well as interaction with screen contents through simplified gestures and navigation.

VoiceOver Overview at Apple.com.

Default Presentation (DP)

Content as presented without Assistive Technologies (review copy and use this term to help distinguish meaning/terms)

Automated Testing

Automated Testing refers to the automatic process of ensuring all app functionality works as intended, free of bugs.

BCP 47 Language Specification

A standard for identifying languages. A BCP 47 language code is a combination of an three/two-letter ISO 639-1 language code and a two-letter ISO 3166-1 region code (e.g. en-US, cpp-ST).

Introduction

Thank you for showing an interest in making your product available to all users, regardless of their disabilities.

The 2011 WHO World Report on Disability estimates more than 15 percent (1 Billion) of the world's population (7 Billion) live with some kind of disability. Contrary to common perception, not all disabilities are severe (e.g. blindness, paraplegia, etc).

The 2010 WHO Global Data on Visual Impairments estimates that an average 30 percent of the population in developed countries suffer from some kind of visual impairment. This includes even light Astigmatism (short/long-sightedness), which in combination with low text contrast may cause uncomfortable headaches.

Insufficient text contrast, bad ergonomics, or high cognitive load impact everyone, it's just that people with severe impairments feel it most, as for them, these issues don't just slow them down, they stop them from achieving their goals.

Everyone can experience External Impairments, they introduce or aggravate existing impairments.

Examples of External Impairments include:

- Bright sunlight may impair color contrast, resulting in temporary color blindness and sight reduction;
- Driving or even walking impair cognitive and motor abilities, resulting in a temporary learning or physical impairment;
- Noisy environments impair hearing, resulting in temporary hearing loss;
- Flashing lights impair vision, causing distraction or epileptic seizures.

Accessibility adds robustness, improving experience for both temporarily and permanently impaired users.

> "Disability need not be an obstacle to success.", (Professor Stephen W. Hawking, 2011 World Report on Disability)

Stephen Hawking is proof that given the right conditions and Assistive Technology, even severe disabilities pose no barrier to living a happy, successful life.

I've put myself in the shoes of someone who relies on Assistive Technology, it didn't take long to encounter a barrier I could not surpass.

We have the moral duty to remove the barriers to participation, to invest time and effort into making tools appropriately accessible to everyone, unlocking their full potential.

“Unfortunately, unseen customers are considered nonexistent.”

The UN Factsheet on Persons with Disabilities estimates that 75% of the FTSE 100 companies in the UK do not meet basic levels of accessibility, thus missing out on more than £96 ($147) million in revenue. Unfortunately, unseen customers are considered nonexistent.

If not for moral values, do it for financial values, either way everyone benefits.

iOS Assistive Technologies

iOS provides free, built-in Assistive Technologies to make apps and websites accessible for people with impairments in the following groups: **Vision**; **Hearing**; **Physical and Motor Skills**; **Learning and Literacy**.

The most popular Assistive Technology is VoiceOver. It enables understanding of, and interaction with on-screen content for vision-impaired users — here's how to enable VoiceOver and give it a try.

Input and Output peripherals (e.g. Switches and Braille Displays respectively), are automatically handled by iOS.

For detailed information, visit Apple's Accessibility Website.

Low Effort—High Reward

iOS Accessibility has been designed to be Low Effort—High Reward.

Low Effort

By design, iOS apps have enough Accessibility Semantics to be **80% accessible**. Reaching **90% is easy** (mostly labeling elements), and **100% is usually trivial**, even for custom elements and gestures.

⚠ **Accessibility isn't a checkbox.** Reaching 100% is greatly appreciated, however, if you can, I'd recommend you create something delightfully accessible (more on that later).

High Reward

Basic Accessibility Semantics enable a wide variety of aids, from Visual (e.g. VoiceOver) to Physical and Motor (e.g. Switch Control). Accessibility Semantics are also the cornerstone of development best practices such as Automated Testing, making Accessibility Conformance a by-product of a robust application.

Creativity

Accessibility on iOS does not limit design possibilities or control over the final interface.

Custom elements and interactions can painlessly be made as accessible as built-in Buttons/Labels/etc (more on that later).

While users control the status of features such as Invert Colors [1] or Mono Audio [2], app creators can and should[3] define how their app behaves when Assistive Technologies are enabled (see documentation and examples for: Accessibility Accommodations, and other Technical Reference topics).

[1] Storyboards are used to define user interface elements of iOS applications.

[2] Mobile Standard Viewing Distance: about 35 cm — Kato, S., Boon, C. S., Fujibayashi, A., Hangai, S., Hamamoto, T. Perceptual Quality of Motion of Video Sequences on Mobile Terminals in Proceedings of the IASTED International Conference, p. 442-7, 2005

[3] Unsafe Red #FF1E1E, safe: #FF4646;
formula: R/(R+ G + B) >= 0.8, and the change in the value of (R-G-B)x320 is > 20 (negative values of (R-G-B)x320 are set to zero) for both transitions. R, G, B values range from 0-1 as specified in "relative luminance" definition. HARDING-BINNIE

Guidelines

Summary

“There was no such document for iOS.”

For the Web, WCAG (Web Content Accessibility Guidelines) is the industry-standard accessibility reference.

WCAG results from the effort of more than 100 accessibility professionals over multiple years, offering solid recommendations, straightforward guideline classification ranging from Single-A (essential) to Triple-A (ideal), and is the adopted legislation by most governments around the world.

There’s no such document for iOS.

Apple offers the best and cheapest (free) assistive technologies, but documentation is either purely technical (#1), in hour-long video form (#2, #3), or consumer-oriented (#4).

This book aims to fill that gap, the guidelines here outlined provide an **actionable** and **easy-to-understand reference** for designers, developers or anyone wanting to understand more about iOS Accessibility.

The four main WCAG principles have been kept (listed below), while references to technologies (e.g. HTML, User-Agent) or input methods (e.g. Keyboard, Mouse) have been removed/replaced with iOS equivalents to ensure validity and familiarity.

The WCAG Accessibility Conformance Levels have been kept, supporting all Single-A Guidelines is required for claiming your app is accessible.

It is understood that fully satisfying higher conformance levels (Double/Triple-A) may not always be possible due to the nature of certain apps (i.e. a drawing app cannot be made fully keyboard-accessible, or a live-streaming app cannot offer accurate real-time captions).

The Accessibility Conformance chapter provides more information on this topic.

Principles

1. Perceivable
2. Operable
3. Understandable
4. Robust

1. Perceivable

“Information can’t be invisible to all of their senses.”

1.1 Text Alternatives

Provide text alternatives for any non-text content.

1.1.1 Non-text Content (Level A)

Guideline

Describe non-text content using Accessibility Semantics so Assistive Technologies can help users perceive (and interact with) content.

Understanding

Assistive Technologies inform and help users interact with your app. For this to happen, your app must be described to Assistive Technologies through Accessibility Semantics.

By understanding your app, Assistive Technologies provide features that allow users to skim content without even seeing the screen, navigate by tilting their heads, reading and writing with a Braille keyboard, and much more without developer having explicitly add support for these and future features.

Examples of non-text content include: Image, Video, Audio, etc; controls such as Sliders, Buttons, Switches, etc; CAPTCHAS, Decoration, Visual Formatting, and Invisible Elements.

Text is an elementary Accessibility Semantic, assigned to elements using an Accessibility Label, it helps the Visual-impaired to at least perceive screen contents through Text-to-Speech.

Solving

Consider and plan how non-text content will be described to users (e.g. Speech) and understood by Assistive Technologies. All non-text content should be described using one or more of the following Accessibility Semantics: Label, Traits, Hint, Value, and Visibility.

Examples

Example #1: An application where the description for a Video element fails to convey its nature or metadata such as duration, is seen as a failure to meet this guideline.

Example #2: An application, such as iOS Photos, where the description for a Video element successfully conveys the element’s nature (Video), metadata useful for element distinction (e.g. duration, creation time), and appropriate Accessibility Semantics such as a Trait identifying the element as interactive (e.g. Button), and a Hint informing the video will be played back upon button activation (double-tapping it). In simpler terms, this application displays videos as buttons with a thumbnail for background, sets their Label to the video's duration and creation date, and Hint with a simple string such as "double-tap to play".

1.2 Time-based Media

Provide alternatives for time-based media.

1.2.1 Audio-only and Video-only (Prerecorded) (Level A)

Guideline

Provide text alternatives containing equivalent information to original prerecorded audio-only and video-only media, so users can still perceive content despite their impairments.

Understanding

Audio-only, video-only, or any content form presented over time, require Hearing, Vision, or Learning abilities that users may not possess.

Assistive Technologies allow users to select alternatives that convey equivalent information through other senses, at a more comfortable pace, or through reading aids.

Solving

Provide Audio Descriptions for the Vision-impaired, Closed Captions for Hearing-impaired, and Transcripts for Learning or Vision & Hearing-impaired.

Refer to Apple's HTTP Live Streaming Documentation for more information, and the HTTP Live Streaming overview for HLS streaming examples and more resources.

Examples

Example #1: An application, where a series of steps are demonstrated in prerecorded video-only media, should provide an audio or text representation of the same content.

Example #2: An application containing prerecorded audio-only media (e.g. Podcasts), should provide a text representation (i.e. transcript).

1.2.2 Captions (Prerecorded) (Level A)

Guideline

Provide Closed Captions for all prerecorded Audio-only, or Audio media synchronized with other formats such as Video, or time-based interactive components. Unless the media is clearly labeled as an alternative for existing text content.

Understanding

Audio media requires Hearing, or Learning abilities that users may not possess.

Solving

Provide Closed Captions so users can perceive content despite their impairments. Alternatively, solutions to Guideline 1.2.3, namely Transcripts, will also ensure content can be perceived despite Hearing or Learning impairments.

Refer to Apple's HTTP Live Streaming Documentation for more information, and the HTTP Live Streaming overview (HLS) for HLS streaming examples and more resources.

Examples

Example #1: An app serving prerecorded audio content (e.g. Podcast) which includes Closed Captions for the selected media.

Example #2: An app serving prerecorded Video content synchronized with an Audio track (e.g. Netflix), which includes Closed Captions or the selected media.

1.2.3 Audio Description or Media Alternative (Prerecorded) (Level A)

Guideline

Provide an audio Description of prerecorded video-only media, an alternative for video-synchronized media (i.e. video + audio) or time-based interactive components (i.e. animation). Unless the media is already a text alternative.

Understanding

Video-only, or any content form presented over time (i.e. animated elements), can only be perceived if Vision is available, or if an individual's Learning & Literacy abilities are sufficient to keep up with the rate of information.

Solving

Provide Audio Descriptions for the Vision-impaired, Closed Captions for Hearing-impaired, and Transcripts for Learning or Vision & Hearing-impaired.

Refer to Apple's HTTP Live Streaming Documentation for more information on how to add audio and video alternatives to media, and the HTTP Live Streaming overview (HLS) for HLS streaming examples and more resources.

No work is necessary if the same information is already conveyed through text, as this provides the use with at least one way to access the information.

Examples

Example #1: An application where the evolution of a cell over a period of time is represented exclusively as video, should also provide an audio description for Vision-impaired users.

Example #2: An application such as Netflix containing prerecorded video media (I.e. Movies, Series, Documentaries), where in addition to Closed Captions, an alternative Audio Description track containing not only the original audio, but additional voiceover describing information contained in the video (e.g. "a woman sits by the window").

Example #3: An educational app containing an interactive animation of a working engine, should also convey the lifecycle through text, which not only can be consumed by Vision-impaired users, but at a pace that's comfortable to the user.

1.3 Adaptable

Enable content to be presented in different ways without losing information or structure.

1.3.1 Info and Relationships (Level A)

Guideline

Provide Accessibility Semantics so Assistive Technologies can help users perceive information, structure, and relationships despite their impairments.

Understanding

At a basic level, Accessibility Semantics allow Assistive Technologies to become aware of, and thus enumerate content.

Enumeration, however, isn't itself enough to convey Information, Structure, or Content Relationships created by visual properties such as color, or relative size and location.

For this reason, it's essential to ensure content can not only be enumerated, but also retain all of the original information.

Solving

Provide sufficient Accessibility Semantics so that Assistive Technologies, and thus users, can fully perceive your app.

Describe:

- Information using Label, Hint, Value;
- Structure using Traits, Visibility;
- Content Relationships using Accessibility Children Grouping;
- Actions using Accessibility Custom Actions.

Examples

Example #1 - Information: An application that describes its visual elements through Accessibility Label, Hint and Value, expressing information that would otherwise need to be extracted from visual properties such as an element's relative location to other elements, spacing, color, and more.

More specifically, a **Label** might be used to describe an icon users cannot see; **Hint** might explain the consequence of tapping that "New" icon which would otherwise made clear by looking at the screen title e.g. "Contacts"; **Value**, complemented by Label and Hint, encapsulates information that is sometimes spread across elements, such as a label containing the human-readable value for an age slider e.g. "31", label and slider would otherwise be related through visual proximity, which users may not be able to see.

Example #2 - Structure: An application where elements are grouped visually by larger text for headers, might also apply a `.Header` Accessibility Trait to convey grouping to Vision-impaired users.

Example #3 - Content Relationships: An application containing two side-by-side elements separated by a divider line, and each element composed of a unit above the unit's count (e.g. Likes and Posts count), related by their close distance. Assistive Technologies cannot parse the visual attributes used to separate elements and unite their components, so the individual components are read as if they're part of a block of text: left-to-right, top-to-bottom (e.g. Likes, Posts, 989, 242). To preserve content relationships and meaning, the application uses `shouldGroupAccessibilityChildren` on each of the two elements. This way Assistive Technologies know that all components within each element should be read before advancing to the next element (e.g. Likes, 989; Posts, 242).

Example #4 - Actions: An application containing a timeline of items, with a range of possible actions for each item. To avoid forcing Vision-impaired users to listen to every available option before advancing to the next item, and infer buttons coming after an item relate to that item. The application hides buttons from Assistive Technologies using `isAccessibilityElement`, and exposes actions through `accessibilityCustomActions` set on the item itself. This way, users can immediately jump to the next/previous item immediately by swiping right/left, be notified of available actions through an auditory cue, browse actions by swiping up/down (selection is always the first action), and activate the desired action by double-tapping.

1.3.2 Meaningful Sequence (Level A)

Guideline

When the sequence in which content is read affects its meaning, use Accessibility Semantics to provide the right sequence.

Understanding

VoiceOver reads (enumerates) all elements individually and sequentially, usually left-to-right and top-to-bottom, like words in a body of text.

But layouts sometimes convey meaning through proximity and other visual properties that VoiceOver does not understand, and users may not be able to see.

Solving

Describe the correct reading sequence using Accessibility Container' `accessibilityElements`, additionally, ensure elements have meaning when read individually by consolidating values and units in a single element using `accessibilityLabel`, hiding the redundant visual label element using `isAccessibilityElement`.

Examples

Example #1: An application containing two side-by-side elements separated by a divider line, and each element composed of a unit above the unit's count (e.g. Likes and Posts count), related by their close distance. Assistive Technologies cannot parse the visual attributes used to separate elements and unite their components, so the individual components are read as if they're part of a block of text: left-to-right, top-to-bottom (e.g. Likes, Posts, 989, 242). To preserve content relationships and meaning, the application uses `shouldGroupAccessibilityChildren` on each of the two elements. This way Assistive Technologies know that all components within each element should be read before advancing to the next element (e.g. Likes, 989; Posts, 242).

Example #2: On the iOS Home Screen, apps composed of two child elements: Icon and Name, are displayed in a grid. Each row is presented to Assistive Technologies as a collection of apps, instead of icons and names, by setting the parent element's `accessibilityNavigationStyle` to `.Combined`, and setting its `accessibilityLabel` to match the Name child element.

1.3.3 Sensory Characteristics (Level A)

Guideline

Instructions for understanding and operating content should not rely solely on Sensory Characteristics as they're incompatible with Vision and Hearing impairments.

Understanding

Vision and Hearing impairments prevent people from perceiving sensory characteristics such as shape, size, visual location, audio, and more.

Relying on a single sensory characteristic will inherently result in content that's harder, or impossible to perceive.

Solving

Use multiple sensory characteristics to increase the chances of content being perceivable despite impairments.

Consider and provide Accessibility Semantics such as Label, Hint, Value that allow Assistive Technologies to help users perceive content through any available sensory characteristics such as Haptics (e.g. text-to-braille).

Use tools such as SimDaltonism to simulate color-blindness, VoiceOver to understand perception in the absence of Vision. Any of these tools help better understand the end product, how it looks and works when used.

Examples

Example #1: The iOS Phone app, where multiple sensory characteristics, namely color and shape, are used to distinguish between unselected and selected state of tab bar items. Deselected items are colored grey and their shape includes only their outline, while selected items are colored blue and their shape changes to a filled version of the same icon. Additionally, `.Selected` is added to the selected tab bar item `accessibilityTraits`, allowing Vision or Hearing-impaired users to either hear the selection state, or read the state description on their Braille keyboards.

Example #2: An application containing a list of items that has just been refreshed, conveys that event through an on-screen message, as well as though an Announcement Notification which will then be converted to the most accessible format for the user (e.g. text-to-speech, Braille).

1.4 Distinguishable

Make it easier for users to see and hear content, including separating foreground from background.

1.4.1 Use of Color (Level A)

Guideline

Color shouldn't be used as the only means of conveying information as it may be partially or completely imperceptible due to impairments.

Understanding

Internal, or External color blindness (i.e. bright sunlight) make color imperceptible.

As a result, any information conveyed solely through color isn't perceived by the user, preventing the app from being used.

Solving

Similarly to the recommendation in "1.3.3 Sensory Characteristics", information shouldn't be conveyed solely through color, use other visual characteristics such as shape, and Accessibility Semantics to ensure information can be perceived despite impairments.

Examples

Example #1: The iOS Phone app, where multiple sensory characteristics, namely color and shape, are used to distinguish between unselected and selected state of tab bar items. Deselected items are colored grey and their shape includes only their outline, while selected items are colored blue and their shape changes to a filled version of the same icon. Additionally, `.Selected` is added to the selected tab bar item `accessibilityTraits`, allowing Vision or Hearing-impaired users to either hear the selection state, or read the state description on their Braille keyboards.

1.4.2 Audio Control (Level A)

Guideline

Provide playback and volume controls for any audio that plays automatically for more than 3 seconds.

Understanding

Auditory stimulus also reduces the ability to process visual information or completing tasks, more so in users with Learning impairments, Internal or External (i.e. ADD or driving, respectively).

Additionally, audio might interfere with Announcement Notifications, relied upon by vision-impaired users, and which may occur at anytime.

For these reasons, untimely or automatically playing audio longer than 3 seconds is deemed disruptive and unacceptable.

Solving

Avoid interrupting ongoing Announcement Notifications by observing `.AnnouncementNotification` and `.AnnouncementDidFinishNotification` and timing audio accordingly. Ensure buttons that trigger audio playback contain Accessibility Traits that reflect that outcome, this way Assistive Technologies know when to silence Announcement Notifications. Refer to documentation on `.StartsMediaSession`, `.UpdatesFrequently`, `.PlaysSound`, and `.AllowsDirectInteraction`.

Automatically playing audio longer than 3 seconds should be timed based on Announcement Notifications, pause control made visible, and accessible through `accessibilityPerformMagicTap`. Additionally, `accessibilityPerformEscape` should also provide an easy way to escape media playback.

Examples

Example #1: An application that schedules its audio output so as not to interfere with Announcement Notifications by pausing audio output when announcements start (`.AnnouncementNotification`), and resuming when they finish (`.AnnouncementDidFinishNotification`). Such application may also specify Accessibility Traits that help Assistive Technologies delay or mute Announcement Notifications, e.g. `.StartsMediaSession`, `.UpdatesFrequently`, `.PlaysSound`, and `.AllowsDirectInteraction`.

Example #2: An application that takes the user to a dedicated Video or Audio playback screen, where visible playback controls are accessible, playback can be paused by performing a Magic Tap gesture, or stopped by leaving the screen via an Escape gesture.

2. Operable

“The interface cannot require interaction that a user cannot perform.”

2.1 VoiceOver Accessible

Make all actions and information available from VoiceOver

2.1.1 VoiceOver (Level A)

Guideline

All functionality is operable through VoiceOver without requiring specific timings, except where the underlying function depends on movement and not just location of user input.

Understanding

Due to Motor or Visual disabilities, users may not be able to meet required input timings, or completely rely on VoiceOver as an input method.

Motor impairments such as Parkinson’s (Internal), or hand muscle/nerve injury (External), prevent users from achieving the level of stability or precision required to perform time-based gestures (i.e. long-press).

Visual impairments difficult eye-hand coordination, making broad (i.e. swipe) or precise gestures (i.e. drag&drop) hard or impossible to perform.

Don’t assume users can see or use the touch screen. Like them, count only on VoiceOver being able to enumerate, announce, and select.

Items are enumerated individually, announced in multiple forms (speech, braille), and selected in multiple ways (tapping anywhere on screen, or even by puffing into a tube).

Solving

All functionality should be operable through VoiceOver.

The exception is where the underlying function depends on movement and not just location of user input (i.e. drawing).

Separate function (i.e. editing, navigation) from trigger (i.e. gestures) and ensure the latter includes VoiceOver.

Assistive Technologies provide you with: Custom Actions, Escape, Magic Tap, and Activation Point.

Examples

Example #1: Activation Point is used to simulate a tap in a specific area within the selected element. I.e. When rearranging the Home Screen and VoiceOver is enabled, tapping anywhere on an app icon activates the delete button positioned on its top left.

Example #2: Activate is used to trigger the same function a gesture would, either because the user cannot perform the gesture or because it conflicts with a built-in VoiceOver gesture such as 2-finger double-tap (Magic Tap), or 2-finger swipe right and left (Escape).

Example #3: Magic Tap is used as a shortcut for the most important function. I.e. In the Phone app it answers and ends phone calls, in the Music app it plays and pauses playback, and in the Camera app it takes a picture.

Example #4: Escape is used as a shortcut for back/close/cancel. Instead of finding and tapping the appropriate button, the user performs a 2-finger swipe right-then-left gesture anywhere on screen.

Example #5: Custom Actions is used to present two or more actions the user can comfortably move through and select. I.e. A timeline-based app (i.e. Twitter, Facebook) could expose actions to be performed on timeline items through Custom Actions (i.e. reply, favorite). Users then navigate between timeline items swiping left/right, between actions swiping up/down, and select the desired action by double tapping anywhere (or another more suitable method).

2.1.2 No VoiceOver Trap (Level A)

Guideline

Any interaction performed with VoiceOver must also be reversible using the same method.

Understanding

VoiceOver might be the only known method for interacting with the app.

Interactions performed with VoiceOver must also be reversible using the same method, otherwise they constitute a trap, frustrating the user and rendering the app useless.

Solving

Don’t rely solely on gestures (i.e. swipe down to close), use them as shortcuts for advanced users but always include standard back/dismiss/cancel buttons with appropriate Label; Avoid reimplementing standard controls such as page view controllers which offer accessible navigation by default; Implement Escape which allows advanced VoiceOver users to escape using a standard 2-finger swipe right-then-left.

Examples

Example #1: Mail allows users to dismiss compose modals using a swipe down gesture, but also displays a standard Cancel button and implements Escape.

2.2 Enough Time

Provide users with enough time to read and use content

2.2.1 Timing (Level A)

Guideline

Avoid time limits, except when essential, with real-time events, or longer than 20 hours.

Understanding

Impairments increase the time users need to complete their goals — the reason they use the app and their primary focus. Time limits that satisfy system requirements, such as expiring sessions, aren't essential to user goals and thus easily forgotten.

This means that changes resulting from expired time limits are likely surprising and prevent users from achieving their goals, either because impairments slow them down, or because they've been forgotten and not taken into account.

As they expire, time limits surprise and frustrate by automatically performing actions that interfere with user goals (i.e. changing content or context).

Solving

Avoid time limits shorter than 20 hours.

The exception is when the time limit is **part of a real-time event**, or when it's **essential to an activity** not viable otherwise.

Examples

Example #1: Auctions constitute a real-time event for which the time limit cannot be changed.

Example #2: 2-Factor Authentication tokens would be invalid as a security mechanism if the time limit is removed.

Example #3: Filling out a form, reading content should not be time-limited.

2.2.2 Pause, Stop, Hide (Level A)

Guideline

For automatically-initiated non-static elements, presented for longer than five seconds and in parallel with other content, there is a mechanism for playback control or hiding which does not invalidate the app when triggered.

Understanding

Automatically-initiated non-static elements include moving, blinking, or auto-updating elements containing text, images or other media.

Blinking content distracts users with attention deficits, and is considered a seizure-inducing flash for users with Photosensitive Epilepsy if bright enough and has a frequency above 3 per second.

Moving or auto-updating content increases reading difficulty for Vision-impaired users.

Solving

Provide a mechanism for playback control or hiding which does not invalidate the app when triggered.

Content not associated with real-time events may be resumed from where it stopped provided the time-shift doesn't convey misleading information.

Avoid moving, auto-updating, and blinking content, never exceed the 3 per second blinking rate.

Examples

Example #1: A full-screen animation isn't presented in parallel, thus excluded from this guideline.

Example #2: Information only valid in real-time such as a stock ticker, a traffic camera, or an auction timer, should be resumed to the most recent information.

Example #3: An explanation animation presented for longer than 5 seconds in parallel with other content should provide a playback control mechanism.

Example #4: An element which blinks for less than 5 seconds to get the user's attention complies with this guideline.

2.3 Seizures

Avoid known seizure triggers

2.3.1 Three Flashes or Below Threshold (Level A)

Guideline

Avoid flashing content more than three times per second or above the general flash and red flash thresholds as these conditions are known seizure triggers.

Understanding

Users with photosensitive seizure disorders can be harmed and prevented from using the app if content meets known seizure triggers.

The most common seizure trigger is content that flashes more than 3 times per second. Seizures become more likely with the following conditions:

- Content size increases beyond than seizure-safe size[4];
- Viewing distance is below the standard viewing distance[5];
- Red color percentage increases[6];
- Luminance goes above the Seizure Luminance Threshold[7].

Solving

Avoid presenting content in a way that matches known trigger criteria.

Examples

2.4 Navigable

Provide ways to help users navigate, find content, and determine where they are

2.4.1 Bypass Blocks (Level A)

Guideline

Provide Accessibility Semantics so VoiceOver users can easily bypass blocks of content.

Understanding

VoiceOver Rotor allows users to discriminate between types of content and navigate a subset of available information. I.e. A user may select “Headings” to navigate between all available Headings and skip other blocks of content such as static text, images, buttons, and more.

[4] Storyboards are used to define user interface elements of iOS applications.

[5] Mobile Standard Viewing Distance: about 35 cm — Kato, S., Boon, C. S., Fujibayashi, A., Hangai, S., Hamamoto, T. Perceptual Quality of Motion of Video Sequences on Mobile Terminals in Proceedings of the IASTED International Conference, p. 442-7, 2005

[6] Unsafe Red #FF1E1E, safe: #FF4646;
formula: R/(R+ G + B) >= 0.8, and the change in the value of (R-G-B)x320 is > 20 (negative values of (R-G-B)x320 are set to zero) for both transitions. R, G, B values range from 0-1 as specified in “relative luminance” definition. HARDING-BINNIE

[7] Seizure Luminance Threshold: 10% or more of the maximum relative luminance where the relative luminance of the darker image is below 0.80

Solving

Built-in iOS components such as buttons or images already contain Accessibility Semantics that distinguish them from other elements.

Ensure other more ambiguous elements, such as built-in Labels or custom views, are appropriately described through Accessibility Traits such as `.Header`, `.StaticText` or `.Link`.

Display text using UITextView and enable Data Detectors for Web Links, Phone Numbers, Events, Addresses, or All.

Examples

Example #1: The iOS Home Screen has a Dock and Main container so that users can quickly jump between these two distinct areas without going through individual apps.

Example #2: The App Store allows users to jump between section headers (Best New Apps, Best New Games) without going through individual apps.

Example #3: Safari allows users to jump between Links on the current page.

2.4.2 Screen Titled (Level A)

Guideline

Screens have titles that describe their topic or purpose.

Understanding

Titles allow users to identify the topic or purpose of a screen, indicating they've found the information they seek, start or resume a task, or simply providing continuity between screens.

In the absence of a title, users are required to identify screens by reading their content or resorting to memory recall.

Identifying screens by reading their content is particularly time-consuming for VoiceOver users as items are read individually, and may not provide enough information to discern screens with similar content (i.e. "Inbox" from "Archive").

Memory recall requires a greater effort (cognitive load) and is error-prone due to External or Internal impairments, such as multitasking or Alzheimer's, respectively.

Solving

Provide a title using the standard Navigation Bar title, Tab Bar Item title or Accessibility Label, or all of the above.

Custom elements containing a `.Header` Accessibility Trait, and described with an Accessibility Label are also acceptable as long as they're the topmost item with that trait (allowing users to swipe upwards until they reach the top of the Headings list accessed via the VoiceOver Rotor).

A Navigation Bar containing only a Search Bar is also acceptable, as long as its Accessibility Semantics provide a unique description.

Examples

Example #1: The App Store Featured screen contains an identifying Navigation Bar title.

Example #2: The Music app Tab Bar Items contain identifying and visible titles.

Example #3: The Instagram app identifies screens using a combination of standard Navigation Bar Titles, sometimes replaced with appropriately described elements (i.e. logo, search bar), and Tab Bar Items that while missing a visible title contain the appropriate Accessibility Label.

2.4.3 Focus Order (Level A)

Guideline

VoiceOver navigation must occur in an order that preserves overall meaning and operability.

Understanding

VoiceOver navigates between individual elements in a left-to-right and top-to-bottom (LTR) sequence, like words in a body of text.

However, VoiceOver isn't capable of understanding reading sequences defined by visual properties such as spacing or color, as a result, VoiceOver users may struggle understanding content meaning or operating the app.

Guideline 1.3.2 ensures elements composed of smaller elements retain their meaning, regardless of reading sequence.

Focus Order differs in the sense that although "LHELO" is a valid sequence of elements (in this case letters), as a whole it doesn't convey the same information as "HELLO".

Solving

Describe the correct reading sequence using Accessibility Container' accessibilityElements, additionally, ensure elements have meaning when read individually by consolidating values and units in a single element using Label, hiding the redundant visual label element using Visibility.

Examples

Example #1: Twitter profile contains the user profile block on the left and profile actions at the same level as the profile photo. Reading order is incorrect as actions are mixed with profile details.

2.4.4 Button Purpose (In Context) (Level A)

Guideline

The purpose of each button should be clear from its Accessibility Semantics alone or in conjunction with its surrounding context.

Understanding

Being able to determine the purpose of a button allows all users to predict its outcome and avoid unintended actions.

In the absence of visual cues due to Vision impairments such as no/low vision, purpose must be described using Accessibility Semantics, which are then translated by VoiceOver into the most accessible form for a given user (i.e. Speech, Braille).

Solving

Clarify purpose through Accessibility Semantics.

For icon buttons lacking any text, buttons with ambiguous text, clarify their purpose by setting an Accessibility Label.

Context also helps clarify purpose, a button labeled “New” is ambiguous by itself, but becomes clearer when located after a Screen Title such as “Contacts”.

For buttons that remain ambiguous despite their label and context, use an Accessibility Hint to explicitly describe their outcome, i.e. “Double-tap to clear notifications”.

Examples

Example #1: The purpose of a button titled “Settings” is made clearer by providing an Accessibility Label such as “Profile Settings”.

Example #2: The purpose of a button titled “Clear”, placed next to a Section Title “Today”, is made clearer by providing an Accessibility Hint such as “Double-tap to clear notifications”.

Example #3: The purpose of a button titled “New” positioned after a Screen Title of “Contacts” is made clear by its context.

3. Understandable

“Content or interaction cannot be beyond their understanding.”

3.1 Readable

Make text content readable and understandable

3.1.1 Language of the App (Level A)

Guideline

Language of the App matches the predominant language in each targeted App Store Region.

Understanding

Users are more likely able to read interface text, data formats, and other characteristics such as interface directionality matching the predominant language of their App Store Region.

Unreadable interface text prevents all users from using the app.

Unfamiliar data formats for date, currency and other units increase the effort required to understand or manipulate content.

Other locale characteristics such as mismatching interface directionality[8] difficult interaction, while symbols, icons and colors may differ in meaning across languages, potentially failing to convey the intended meaning.

Solving

Ensure visible and invisible user interface text, such as screen titles and Accessibility Semantics, are available in the predominant language of each targeted App Store Region. Use NSLocalizedString to distinguish user interface text, and for iOS to display the appropriate translation.

Media such as images, audio, video, text alternatives such as captions, should also be localized.

Use NSFormatters (e.g. Number, Date, ByteCount, Energy, Mass, Length, Distance, more) for iOS to automatically display data in a familiar format.

Use standard components, such as Navigation Controllers or Sliders, for iOS to automatically change directionality according to the user’s language direction.

[8]Storyboards are used to define user interface elements of iOS applications.

Examples

Example #1: A utility app such as Maps should have its title localized to “Mapas” for users in the Portuguese App Store Region.

Example #2: A public transport app targeted at US users uses distance formatters so that distance is automatically displayed in Kilometers for visiting non-US users.

Example #3: The Mail app uses standard Navigation Controllers which automatically adjust to the user’s language directionality conventions. E.g. For Chinese users, left and right navigation bar buttons swap positions, and views slide in from the left instead of right as it happens in left-to-right languages such as English. Additionally, Leading and Trailing Auto Layout Constraints are swapped for right-to-left languages.

Example #4: An app replaces the color Red with White for destructive actions for the Chinese App Store Region to preserve meaning.

Example #5: An app replaces a green four leaf clover with a red luck symbol for the Chinese App Store Region to preserve meaning.

Example #6: An app horizontally flips an icon representing a list for a right-to-left language user as the representation contains references to left-to-right languages.

3.2 Predictable

Make apps appear and operate on predictable ways

3.2.1 On Focus (Level A)

Guideline

When any component receives focus, it does not initiate a change of context.

Understanding

Components receive focus as users navigate screen contents using VoiceOver.

The focused element is highlighted using an Accessibility Frame, and inspected by VoiceOver for any Accessibility Semantics that can be used to identify the element and possible actions.

Receiving focus does not express user intent.

Changing context by changing screen contents or performing navigation when an element is focused will interrupt VoiceOver Navigation, leading to surprise and confusion.

Solving

Do not react to VoiceOver Focus events.

The exception is when the default interface mimics the VoiceOver Focus mechanism by requiring users to focus an element before selecting it. In this situation, the underlying focus mechanism should be synchronized with VoiceOver Focus.

Examples

Example #1: Moving focus between filters in a photo editing application does not apply them.

Example #2: When creating a new even in Calendar.app, focusing on the start date does not expand the date picker.

3.2.2 On Input (Level A)

Guideline

User input changes context only when the user has been advised of the behavior.

Understanding

User input includes: text entry, toggling a switch control, selecting an option from a list.

It does not include: performing a gesture, activating a button.

Change of context includes: content is changed in a way that alters its meaning; navigating to a new screen.

It does not include: expanding a component (unless its content becomes focused); filtering or sorting content; changing minora visual attributes.

Users do not expect to be taken out of the current context, when not explicitly requested.

Doing so will shock users with Learning impairments, preventing them from using the app; unexpectedly end the current task causing frustration to all users.

Solving

Only change context as a result of clear user intent such as selecting content (i.e. Email), or pressing a button (i.e. Login).

Examples

Example #1: An application that moves focus between phone number fields based as user inputs their values isn't considered a failure as neither form meaning or task is interrupted. Input should however behave as a single field in regards to backspace, or other operations that would fail due to the automatic change of focus.

Example #2: An application that automatically attempts a login upon detecting user input in both username and password fields is considered a failure to meet this guideline as that consequence wasn't clearly communicated or required by the user.

Example #3: An application where the first two items of a segmented controller change sorting order, while the third item takes the user to a new screen is considered a failure, unless the third option makes this consequence very clear.

3.3 Input Assistance

Help users avoid and correct mistakes

3.3.1 Error Identification (Level A)

Guideline

Input errors should be communicated through text.

Understanding

Users may not perceive sensory characteristics such as color due to low/no vision or color blindness (both potentially caused by External Impairments i.e. bright sunlight), Hearing and Learning impairments pose similar barriers to audio cues, or the understanding of icons.

Communicating through Text, instead of the aforementioned sensory characteristics, ensures all users can become aware of, and correct errors.

Solving

Make users aware of errors through UIAlertControllers for critical errors, or Announcement Notification for non-blocking warnings, allowing VoiceOver to convert the text into Speech, Braille, or another appropriate form.

Use Accessibility Labels to provide accessible error descriptions (i.e. “Input with error, Email, Gmail emails aren’t currently supported.”).

Prevent input errors through good form design: limiting freeform user input when only a limited set of options is allowed by providing Sliders or Pickers, clearly labeled inputs and Accessibility Hint describing acceptable parameters.

Examples

Example #1: An messaging application that informs users of missing network connectivity using a `UIAlertController` or Announcement Notification upon the user attempting to send a message.

Example #2: A social networking application that failing to reload new information due to network issues informs the user by displaying the error as text and posting a `UIAccessibilityScreenChangedNotification` so that VoiceOver is aware of the change and reads the new element.

Example #3: A signup form that informs the user of an incorrect email through a `UIAlertController`, and subsequently moves VoiceOver Focus into the Email field described as “Input with error, Email, Gmail emails aren’t currently supported”.

3.3.2 Labels or Instructions (Level A)

Guideline

Labels or instructions are provided as text when content requires user input.

Understanding

Users need to understand what's required of them and how to provide that information.

Using sensory characteristics such as color (i.e. red) or symbols (i.e. ”*”) to denote mandatory may pose barriers to Vision or Learning-impaired users.

Solving

Use Accessibility Label and Hint to help users identify and operate user input controls.

When providing clarification through static text such as a UILabel or Section Header, ensure relationship to user input control is clear as per 1.3.1 Info and Relationships, and content isn't duplicated by Accessibility Semantics (Label/Hint) as per 1.3.2 Meaningful Sequence.

Use built-in user input controls users are familiar with and already know how to operate without extra clarification.

Provide just enough information to meet this criteria, too much information or instruction is as harmful as too little.

Whenever possible, do the heavy lifting for the user.

Information such as Age or Weight can be fetched from HealthKit; ACAccountStore provides instant access to Facebook, Twitter and other social accounts; Apple Pay provides not only payment details but also Name, Email, Phone and Address for the device owner, while Contacts provide the same and other info for user contacts; personal event information can be retrieved and saved to Calendar; Reminders can be read and saved to Reminders; Camera can be used to retrieve long Credit Card numbers, facial recognition or machine-readable information; Photos can provide the latest photo taken or complex queries for retrieving photos for a given date or location, Core Location can be used to fetch the current address or components such as post code.

User information can be saved and shared between apps and across installs using CloudKit, App Groups and Keychain, so the user needs to enter data only once.

Examples

Example #1: The iOS Calendar New Event form describes the “Start Date” field using a Label which includes the field name and current value, followed by a Hint that informs the user the value can be edited by double-tapping.

Example #2: An application that allows the user to pick from 3 different options, selected option can be change by swiping up and down, with selection being announced and described, the same description is duplicated by a label that can be reached by navigating away from the picker, this represent a failure as the duplicate information might confuse VoiceOver users.

Example #3: An application that uses built-in components such as Text Fields, Sliders, and other, is immediately understood by the user without any developer effort.

Example #4: An application that completely removes the need of manual address input by pulling it from Apple Pay.

Example #5: An application that's been reinstalled retrieves previous user settings from CloudKit, removing the need to enter them again.

4. Robust

“As technologies evolve, the app should remain accessible.”

4.1 Compatible

Maximize compatibility with current and future Assistive Technologies

4.1.1 Native and Future-Proof (Level A)

Guideline

Applications should be made accessible using Native Assistive Technologies, and future-proofed.

Understanding

iOS Native Assistive Technologies offer a free, mature solution with around 150 features catering for users with Vision, Hearing, Physical & Motor Skills, and Learning & Literacy impairments.

Third-party input and output hardware, such as braille keyboards, hearing aids, switches, work out-of-the-box with zero developer effort.

Assistive Technologies are enabled by Accessibility Semantics, already included in built-in components such as Navigation Controllers, Labels, Buttons, Sliders, and more provided by User Interface frameworks (UIKit).

By using Native User Interface Frameworks and Assistive Technologies, iOS apps can be made Accessible in a low-effort/high-reward fashion, with the assurance new and improved Accessibility Features are automatically supported when released.

Solving

Designers should consider the implications of using custom components, balancing their value with developer effort required to make them accessible, and the learning curve for all users.

Custom components are not discouraged but the required Accessibility Semantics should be designed as well, which Trait should the component have? How will it be announced? How will users interact with it?

Developers should follow best practices such as feature availability checks to ensure code is robust enough to allow the usage of new features, without introducing bugs to devices where such feature is not supported.

Examples

Example #1: An application released for a previous major OS version, implemented using Native UI Frameworks, allowing users of a newly-released OS version to take advantage of a new character-by-character spelling feature without modification to the app.

Example #2: An application containing a custom component that allows all users to swipe right to 'like' an item, and allows VoiceOver users to perform the same action using Accessibility Actions which allow users to iterate through Actions by swiping vertically and double-tapping to select the desired action.

Example #3: An application that takes advantage of an Accessibility Feature introduced by the latest operating system release, and retains backwards-compatibility through feature detection.

4.1.2 Accessibility Semantics (Level A)

Guideline

Custom user interface components contain enough Accessibility Semantics to make them readable and operable.

Understanding

Built-in UI Components save you from reinventing buttons or navigation, and providing them with basic Accessibility Semantics.

Custom UI Components, such as a graphs or gestures, are created from scratch and lack Accessibility Semantics by default.

Solving

Consider/Design how Custom UI Components will be announced and interacted with through VoiceOver.

Implement Custom UI Components and their Accessibility Semantics so they pass the iOS Accessibility Semantics Audit, ensuring Purpose, Name, Personality, Value, Operation, Location.

Examples

Example #1, Purpose: Given an Accessibility Value allows VoiceOver to announce the precise value of a component, such as a slider. A visual label containing the same value is redundant and like decorative components, can be hidden through isAccessibilityElement.

Example #2, Name: Accessibility Label allows VoiceOver users to identify a component. A label such as "Play Bad Girls by M.I.A." can be used in conjunction with a `.Button` Accessibility Trait, so users perceive an element's interactive nature and outcome.

Example #3, Personality: Accessibility Traits define a component's personality. VoiceOver users rely on Traits to know when and how to interact with components such as Buttons; browse components with specific personalities such as Headers bypassing irrelevant information (2.4.1 Bypass Blocks); convey Selected/Enabled states, and more.

Example #4: Value: Accessibility Value allows VoiceOver to announce the value of a component which may be adjustable or change over time.

Example #5, Operate: An Accessibility Hint is used to inform users how to operate a component. Built-in components such as Buttons or Sliders provide default Hints such as "double-tap to open" (Buttons) or "swipe up or down with one finger to adjust" (Sliders). Custom interactions, such as accessing Drafts by long-pressing the compose email button in Apple Mail, are explained through hints.

Example #6, Location: VoiceOver highlights the currently focused item using an Accessibility Frame, allowing users to infer meaning and purpose from surrounding items, and recovering from attention or memory issues.

Conformance

For the purpose of Accessibility Conformance, Web Accessibility Guidelines 2.0 (WCAG 2.0) have been adopted or adapted by the vast majority of governments, including Europe, Australia, New Zealand, or Israel.

In the United States, Section 508 of the Rehabilitation Act of 1973, a subset of WCAG 2.0, is used as the official legislation.

Accessibility Conformance makes business sense, as it increases your user base, and is mandatory for products used by Government entities, NGOs, Education, and industries such as Travel. This will change depending on country, and have in mind lawsuits against inaccessible products are frequent.

The guidelines included in this book are based on WCAG 2.0, while references to web and desktop have been replaced with their iOS equivalents, the original goals have been respected to ensure Accessibility Conformance.

Accessibility Conformance validation is a manual process as there are currently no automated tools for this purpose.

To aid you with Accessibility Conformance validation I've included in this book a section titled "Accessibility Semantics Audit" (ASA).

Additionally, if you're targeting the US market, the Voluntary Product Accessibility Template® (VPAT®) will help you document your product's conformance with Section 508. Within VPAT, the section you're looking is "Section 1194.21 Software Applications and Operating Systems".

IMPORTANT: No tool or process will provide a complete guarantee your product is Accessible to everyone, even if it conforms to all Accessibility Guidelines, due to the unpredictable nature of impairments, users may still encounter challenges when using your product. Conversely, a product that does not comply to some Accessibility Guidelines may still be Accessible as the supported guidelines may account for all the needs of a specific user.

Accessibility Semantics Audit

The Accessibility Semantics Audit (ASA) helps quickly determine a basic level of Accessibility Conformance, below Single-A, for individual user interface elements.

While Single-A is required to conform with Government legislation, ASA-level conformance still provides significant value at a very low cost.

The value of ASA lies in the effort to reward ratio. First, the simple querying of every user interface element with 6 straightforward questions (e.g. Where am I?), results in a preliminary Accessibility Compliance status.

Second, applications created with built-in user interface components leave few questions unanswered, filling in the gaps is not only easy, but also goes a long way to meeting many of the Accessibility Guidelines which only require the presence of Accessibility Semantics, and facilitates Automated Testing.

The following is a list of questions, along with their description and answer information.

- Is it relevant for Accessibility?
- Is it succinctly and accurately identified?
- Are Behavior, Impact, Type and State perceivable?
- Is its value accurate at any point in time?
- Are outcome and interaction understood?
- Where is it?

Is it relevant for Accessibility?

Discussion

Purely visual elements: Separator lines are visually important (e.g. two line-separated values). However, if elements are expressed as separate entities on a Storyboard[9] level, the separator line provides no value, as Assistive Technologies such as VoiceOver have no trouble understanding and conveying to the user the separation expressed in the Storyboard.

Redundant information: Labels containing the current value of a slider are visually important (e.g. age slider and its label). However, ATs are able to announce elements and their value as a single element, making the label redundant.

Answering

Toggle visibility using `isAccessibilityElement: Bool`. A negative value would cause Assistive Technologies (e.g. VoiceOver) to completely ignore an element.

[9] Storyboards are used to define user interface elements of iOS applications.

Is it succinctly and accurately identified?

Discussion

Absent Identity: For text content (e.g. labels) identity can be derived from its text, the same isn't true for non-text content (e.g. icon buttons, graphs, more in "1.1.1 Non-text Content"). Elements lacking identity cannot be considered accessible as they cannot be perceived by users.

Verbose Identity: Verbosity increases the time required to perceive and navigate elements, thus slowing and frustrating users. A verbose description such as "Create a new list" should be shortened to "Create", as its outcome can be explicitly set through `accessibilityHint` (see "Are outcome and interaction understood?") or derived from its context.

An element's identity should be absent of usage instructions or value as those are better conveyed through Personality/Interaction and Value, respectively.

To ensure applications can be used in an efficient manner, elements should contain succinct, yet accurate descriptions.

Inaccurate Identity: Limitations in Assistive Technologies may lead to inaccurate interpretations of an element's identity as described through text. ATs lack the cultural knowledge that allows us to discern a single "dollar sign" from "cheap", or "eighteen forward slash one" from "January 18th". Remember to cover for lack of cultural knowledge by providing AT-friendly textual descriptions.

Answering

Define an element's identity using `accessibilityLabel: String?` taking the discussion topics into account.

Content skimming speed can be improved by placing unique information at the start of the copy (e.g. "#1 Account, #2 Account" instead of "Account #1, Account #2"), this allows users to extract essential information just by reading/listening to the first characters of an element.

Are Behavior, Impact, Type and State perceivable?

Discussion

The personality of a component is the combination of one or more `UIAccessibility-Trait`: Behavior (e.g. Adjustable), Impact (e.g. PlaysSound), Type (e.g. Header), State (e.g. Selected).

- Behavior: Provides interaction affordances (e.g. Buttons can be double-tapped), and allows ATs to control custom elements (i.e. custom sliders);

- Impact: Improves app integration for ATs (e.g. avoid interfering with media playback);
- Type: Allows content skimming by enabling Type filtering (e.g. navigate through Headers-only);
- State: Informs users about the state of an element (e.g. Option 1 is selected).

Answering

Set traits using `accessibilityTraits: UIAccessibilityTraits`, so that Behavior, Impact, Type, and State (BITS) are accurately described.

Is its value accurate at any point in time?

Discussion

While `accessibilityLabel` describes an element's identity, `accessibilityValue` describes an element's value, even as it changes over time (i.e. a clock changes over time, sliders change with user interaction).

Answering

When value is absent or read literally (e.g. "9:41" as "Nine colon Fourty-One"), set value using `accessibilityValue: String?`.

The `NSNumberFormatterSpellOutStyle` makes spelling out numbers units trivial.

NOTE: Labels used to display values are made redundant by `accessibilityValue`, ensure those are hidden from ATs through `isAccessibilityElement`.

Are outcome and interaction understood?

Discussion

Outcome: Certain interactive components, despite having clear Identity and Behavior, may not hint at their outcome (e.g. A "New" button does not reveal what it creates, a slider may not reveal the adjusted variable).

Interaction: Built-in components (e.g. Buttons, Sliders) offer generic hints for familiar behaviors (e.g. "Swipe up or down to adjust the value", "Double-tap to open"). Hints should be provided for custom interactions on either built-in or custom components, e.g. "Double-tap and hold to view Drafts" and "Double-tap and hold, wait for a sound, drag to reorder", respectively.

Answering

Explain outcome or interaction through `accessibilityHint: String?`.

Where is it?

Discussion

Certain ATs such as VoiceOver and Switch Control, highlight the element currently focused by their navigation mechanism, this helps users with non-severe vision impairments disambiguate and extract information from an element's spatial context.

Answering

For the vast majority of elements, ATs use the frame to draw the highlight cursor shape, if necessary, a custom shape can be provided through `accessibilityFrame: CGRect` or `accessibilityPath: UIBezierPath?` (for irregularly-shaped elements such as the ones commonly found in games).

Technical Reference

Intro

This section includes main and less known accessibility-related methods, constants, notifications and helper functions from the iOS SDK (9.0).

This documentation has been made easier to understand by reducing the amount of abstraction, jargon, verbosity, while still including declaration information, usage tips and examples.

Main takeaways:

- All UIViews conform to the `UIAccessibility` protocol
- Built-in iOS technologies do most of the work for you
- Accessibility APIs provide granular control, down to speech language, pitch, and interpretation.

The `UIAccessibility` prefix in methods, variables or classes has been collapsed to a single period character (".") to speed up text skimming. E.g. `.IsBoldTextEnabled` equates to `UIAccessibilityIsBoldTextEnabled`.

UIAccessibility

This is an informal protocol all UIKit UIViews conform to providing the basic Accessibility API.

Many of its methods derive their value from UIKit elements, requiring none or small application-specific details when values are incomplete.

Values can be set or refined either programmatically or using Interface Builder.

`accessibilityActivationPoint`

Overrides the actual UITouch location coordinates for the purpose of aiding users to accurately hit small screen elements.

Declaration

```
var accessibilityActivationPoint: CGPoint
```

Discussion

Defaults to the midpoint of the `accessibilityFrame`.

Examples

Example #1: The iOS Home screen app icon activation point is usually its center, but while in edit mode, tapping anywhere in the app icon area will activate the delete button located in the upper right corner.

`accessibilityElementsHidden`

Makes all children of the receiving element invisible to ATs.

Declaration

```
var accessibilityElementsHidden: Bool
```

Discussion

Defaults to `false`. Commonly used to hide elements that are not the focus of the current action. Shortcut to manually setting `isAccessibilityElement` to `false` on all elements inside a parent element.

`accessibilityFrame`

Coordinates and rectangle used by ATs to highlight a focused element.

Declaration

```
var accessibilityFrame: CGRect
```

Discussion

Defaults to using the frame of the `UIView` (or subclass of), `CGRectZero` for non-UIView elements. This value should be updated if the frame changes after the view is initialized.

`accessibilityHint`

Explain interaction or its outcome.

Declaration

```
var accessibilityHint: String?
```

Discussion

Defaults to system-provided hint for UIKit controls, `nil` for custom controls.

`accessibilityLabel`

Describe an element.

Declaration

```
var accessibilityLabel: String?
```

Discussion

Defaults to a value derived from a UIKit element's title, `nil` for custom elements.

`accessibilityLanguage`

Set an element's language, impacting text-to-speech outcome.

Declaration

```
var accessibilityLanguage: String?
```

Discussion

Defaults to the user's current language, set using a language ID tag following the BCP 47 standard.

`accessibilityNavigationStyle`

Causes Assistive Technologies to ignore child elements and read only the receiver's `accessibilityLabel`.

Declaration

```
var accessibilityNavigationStyle: UIAccessibilityNavigationStyle
```

Discussion

Defaults to `UIAccessibilityNavigationStyle.Automatic`. Use to combine complex child view hierarchies into a single parent element (the receiver). Information contained in child views should be available on the parent via an `accessibilityLabel`, and any actions exposed through `accessibilityCustomActions`.

`UIAccessibilityNavigationStyle`

The navigation style.

Declaration

```
enum UIAccessibilityNavigationStyle : Int {
    // Allow Assistive Technologies to decide how element should be \
read, usually .Separate
    case Automatic
    // Read the receiver's elements individually
    case Separate
    // Read only the receiver
    case Combined
}
```

Examples

Example #1: On the iOS Home Screen, apps composed of two child elements: Icon and Name, are presented to Assistive Technologies as a single element by setting the navigation style of the parent element to `.Combined`, and setting its `accessibilityLabel` to match the Name child element.

`accessibilityTraits`

One or more `UIAccessibilityTraits` characterizing Behavior, Outcome, Type and State of an element.

Declaration ‘

‘{lang="swift"} var accessibilityTraits: UIAccessibilityTraits

Discussion

Defaults to traits inherited from the UIKit element (or subclass of), `UIAccessibility-TraitNone` for non-UIView elements.

Built-in elements are already characterized with the most appropriate traits, e.g. Buttons already include Button, Sliders already include Adjustable. Custom elements require you to select the traits that best characterize them.

The following Accessibility Traits exist:

> The period denotes a `UIAccessibilityTrait*` prefix, e.g. `.Button` corresponds to `UIAccessibilityTraitButton`.

- **Behavior**
  - **.Button**: The element can be activated by the user, either through a single, or double-tap if using VoiceOver. Calls the `accessibilityActivate()` method on Non-UIKit elements (e.g. `UIAccessibilityElement`).
  - **.Link**: The element can be activated by the user, just as a Button, but provides extra semantic context, enabling content skimming and informing the user activation may result in leaving the current app.
  - **.SearchField**: The element can be activated by the user, just as a Button, but provides extra semantic context, enabling content skimming and informing the user activation will require input.
  - **.KeyboardKey**: Allows users to chose faster typing modes that bypass standard announcement and activation behavior (i.e. double-tap to activate). By using VoiceOver Rotor, Typing Mode can be changed to
    * **Touch Typing**: activation occurs when releasing the finger from the screen, announcements occur as usual (i.e. on finger press);
    * **Direct Touch Typing**: acativation occurs on finger press, announcements occur after activation for feedback.
  - **.Adjustable**: The element represents a value that can be adjusted by swiping up or down, calling `accessibilityIncrement` or `accessibilityDecrement` on the receiver, respectively.
  - **.AllowsDirectInteraction**: The element allows direct interaction, as if Assistive Technologies were disabled. Useful for elements such as piano keyboard.
  - **.NotEnabled**: The element is not enabled and does not respond to user interaction.
- **Outcome**
  - **.PlaysSound**: The element plays a sound when activated. Allows Assistive Technologies to avoid playing Announcement Notifications simultaneously with app audio.
  - **.UpdatesFrequently**: The element frequently updates its label or value (e.g. stopwatch). To avoid overwhelming the user, Assistive Technologies will only announce updated content on demand.

  - **.StartsMediaSession**: The element starts a media session (i.e. audio/video playback or recording). ATs will remain silent upon element activation to avoid interference.
  - **.CausesPageTurn**: The element is one in a series of pages. When element content has been read, Assistive Technologies will call `accessibilityScroll` on the element with `UIAccessibilityScrollDirectionNext` as argument. This informs the app to perform a page turn and present new content to the user, scrolling will stop when content no longer changes.
- **Type**
  - **.Header**: Elements of this type will have the word "Heading" appended to their Accessibility Label, users can skim content by locking navigation to this type through VoiceOver Rotor.
  - **.Image**: Elements of this type will have the word "Image" appended to their Accessibility Label. Use with images that contain important information, can be combined with the `Button`' trait for interactive images.
  - **.StaticText**: Provides Accessibility Semantics indicating the element cannot be interacted with and presents only information that can be read.
  - **.SummaryElement**: Provides a summary of the screen, ATs will read this element automatically when the screen is presented.
- **State**
  - **.Selected**: The element is announced as being in a selected state. Commonly used with controls such as Switches, Table Cells, Tab Bar Items, Segmented Control Segments, and more.
- **Other**
  - **.None**: The element has no traits.

`accessibilityPath`

Used to highlight elements with non-rectangular shapes, overrides `accessibilityFrame`.

Declaration

```
@NSCopying var accessibilityPath: UIBezierPath?
```

Discussion

Defaults to `nil` and `accessibilityFrame` is used instead.

`accessibilityValue`

Used to inform the user about the current value of an element, potentially in a friendlier language.

Declaration

```
var accessibilityValue: String?
```

Discussion

Defaults to a value derived from a UIKit Control' current value (i.e. Slider), `nil` for other elements (e.g. custom control).

Examples

Example #1: The `accessibilityValue` for a "Price" UIKit Slider set to "50%" will be the directly derived value of "50%", effectively announced to the user as "Price: 50%" (Label: Value). While this value is accurate, a better user experience would be to use the more meaningful computed value (i.e. 35 USD/GBP) likely being used on a related label.

Example #2: A custom control (e.g. Input Field, Button, Content Item, Navigation Item) may have an identifying label (e.g. "Message text, Input Field", "Buy, Button", "From Jane at 09:41", "Messages, Tab Bar Item"), but lack the a descriptive value (e.g. "Message text, Input Field, Hello there", "Buy, Button, 35 USD/GBP using Visa ending in 5301", "From Jane at 09:41, Good morning!", "Messages, Tab Bar Item, 3 Unread").

`accessibilityViewIsModal`

Causes Assistive Technologies to ignore sibling views within the same parent view.

Declaration

```
var accessibilityViewIsModal: Bool
```

Discussion

Defaults to `false`. Setting this to `true` causes Assistive Technologies to ignore all other views contained within the same parent view.

Examples

Example #1: A window containing View A and View B, where A is modal, causes ATs to ignore B. Additionally, if a child View B.1 is set to modal, that does not cause View A to be ignored as their parent view differs.

`isAccessibilityElement`

Toggles whether an element is visible to Assistive Technologies.

Declaration

```
var isAccessibilityElement: Bool
```

Discussion

Defaults to `true` for UIKit elements, `false` for everything else. Elements not relevant to Accessibility users should be marked with `false` (see: Is it relevant for Accessibility?).

`shouldGroupAccessibilityChildren`

Changes the navigation flow Assistive Technologies.

Declaration

```
var shouldGroupAccessibilityChildren: Bool
```

Discussion

Defaults to `false`, set to true for Assistive Technologies to change their navigation flow so that children of the receiving View A are read in sequence before moving onto the sibling View B. Helpful to ensure conformance with guidelines "1.3.1 Info and Relationships" and "1.3.2 Meaningful Sequence".

This property differs from `UINavigationStyle.Combined` in the sense children are still read individually, whereas `.Combined` combines children and reads only the top element.

Examples

Example #1: Consider a screen containing Views A, B, and C, each with two vertically-stacked children for displaying a Unit and its Value. With `shouldGroupAccessibilityChildren` set to `false` (the default) on views A, B, and C, the reading sequence would occur in the default left-to-right, top-to-bottom fashion, rendering the content hard to understand (i.e. Age, Likes, Followers, 31, 24K, 9001). Setting `shouldGroupAccessibilityChildren` to `true` on Views A, B, and C would result in a meaningful reading sequence: Age: 31, Likes: 24K, Followers: 9001.

`UIAccessibilityContainer`

Provides a way make non-UIView children of UIView elements accessible (e.g. elements of a data visualization).

This is an informal protocol all UIViews conform to.

⚠ The container UIView element should be hidden from ATs by setting `isAccessibilityElement: false`, as it doesn't provide any purpose for Accessibility Semantics.

`accessibilityElementAtIndex`

Returns the `UIAccessibilityElement` at the specified index.

Declaration

```
func accessibilityElementAtIndex(_ index: Int) -> AnyObject?
```

Discussion

Defaults to `nil`.

`accessibilityElementCount`

The number of `UIAccessibilityElement` elements in the container.

Declaration

```
func accessibilityElementCount() -> Int
```

Discussion

Defaults to `0`.

`accessibilityElements`

Returns an array of `UIAccessibilityElement` elements.

Declaration

```
var accessibilityElements: [AnyObject]()?
```

Discussion

Defaults to `nil`. Used to expose the content of non-UIView elements to Assistive Technologies.

`indexOfAccessibilityElement`

Returns the index of the specified `UIAcessibilityElement`.

Declaration

```
func indexOfAccessibilityElement(_ element: AnyObject) -> Int
```

Discussion

Defaults to `NSNotFound`.

`UIAccessibilityElement`

Encapsulates Accessibility Semantics for elements within a `UIAccessibilityContainer`.

Declaration

```
init(accessibilityContainer: AnyObject)
```

Discussion

A `UIAccessibilityElement` provides access to all properties and methods of the `UIAccessibility` protocol, adding the following property: `var accessibilityContainer: AnyObject?`.

`accessibilityContainer`

Provides access to the `UIAccessibilityContainer` of a `UIAccessibilityElement`.

Declaration

```
var accessibilityContainer: AnyObject?
```

Discussion

Defaults to `.None`. This property is only available on `UIAccessibilityElement` views.

UIAccessibilityAction

Provides a way to define accessible triggers, shortcuts, and additional behavior for actions performed through Assistive Technologies.

This is an informal protocol all UIViews conform to.

Accessible Triggers: because not all users can perform complex or any gestures at all, Assistive Technologies offer a simplified interaction vocabulary: double-tap to activate, 3-finger to scroll, swipe left or right to navigate between screen elements, and swipe up or down to navigate contextual options (e.g. element actions, text characters, adjustment values, on-screen elements of a specific type).

Switch Control offers an extremely simplified interaction mechanism whereby through a single switch, users can navigate and select actions presented in a contextual menu. This Switch Control Menu is populated with all actions available to the user, from system-wide scrolling and 3D Touch, to element-specific actions such as archiving (see: `accessibilityCustomActions`).

Shortcuts: The structure used to separate content from actions provides: accessible triggers, a predictable method for browsing and selecting contextual options (element/screen), the possibility of including accessibility-specific actions not present in the Default Presentation, and finally, system-wide shortcuts such as Magic Tap (a 3-finger double-tap dedicated to performing the most common action on screen), and Escape (2-finger swipe right then left dedicated to escape the current context).

Additional Behavior: However actions are selected, in addition to built-in handlers (e.g. `IBAction`, `scrollViewDidScroll`), `UIAccessibilityAction` provides additional handlers such as `accessibilityScroll` allowing for accessibility-specific behavior.

`accessibilityActivate`

Handles and allows programatic element activation.

Declaration

```
func accessibilityActivate() -> Bool
```

Discussion

Elements can be activated by performing triggers, these include gestures (e.g. tap, swipe up), shaking, and more. This accessibility method provides a programatic trigger, as well as a way to specify additional behavior to be performed after non-accessibility handlers.

`accessibilityCustomActions`

Provides a way for exposing actions to Assistive Technologies, separating them from content.

Declaration

```
var accessibilityCustomActions: [UIAccessibilityCustomAction]?
```

Discussion

Specifying actions through this method separates them from content, enabling the use of system-wide, predictable, and efficient methods of browsing content or actions.

`accessibilityDecrement`

Called when the value of an `.Adjustable` element is decremented.

Declaration

```
func accessibilityDecrement()
```

Discussion

Called after any existing handlers.

`accessibilityIncrement`

Called when the value of an `.Adjustable` element is incremented.

Declaration

```
func accessibilityIncrement()
```

Discussion

Called after any existing handlers.

`accessibilityPerformEscape`

Handles and allows programatic triggering of the Escape action.

Declaration

```
func accessibilityPerformEscape() -> Bool
```

Discussion

Escape is a predictable, system-wide method for escaping the current context (i.e. screen or state). It can be triggered by performing a 2-finger swipe right-then-left gesture or via the Switch Control Menu. Success should be reported by returning `true`.

`accessibilityPerformMagicTap`

Handles and allows programatic triggering of the Magic Tap action.

Declaration

```
func accessibilityPerformMagicTap() -> Bool
```

Discussion

Magic Tap is a predictable, system-wide method for performing the current context's main action (e.g. answer/hangup a call, pause/resume audio, submit form). It can be triggered by performing a 2-finger double-tap gesture or via the Switch Control Menu. Success should be reported by returning `true`.

`accessibilityScroll`

Handles and allows for programatic scrolling.

Declaration

```
func accessibilityScroll(_ direction: UIAccessibilityScrollDirection\
) -> Bool
```

Discussion

Called after any existing scroll handlers (e.g. `scrollViewDidScroll`). Usage examples include providing additional feedback by posting a `UIAccessibilityPageScrolledNotification` with the new page status as a parameter (e.g. "Page 3 of 9").

`UIAccessibilityScrollDirection`

The scrolling direction.

Declaration

```
enum UIAccessibilityScrollDirection : Int {
    case Right
    case Left
    case Up
    case Down
    case Next
    case Previous
}
```

Discussion

Values are directional (Up, Down, Left, Right) or sequential (Next, Previous).

UIAccessibilityCustomAction

Represents a custom action to be provided to `UIAccessibilityAction`'s `accessibilityCustomActions` method.

Declaration

```
init(name name: String,`
` target target: AnyObject?,
 selector selector: Selector)
```

- **name**: A localized short and descriptive name for the action, usually the same used for the button label if one exists (e.g. “Archive”, “Favorite”).
- **target**: The object containing the selector.
- **selector**: The selector within *target* to be used as a handler. Possible values include:
  - `func mySelector -> Bool`
  - `func mySelector(action: UIAccessibilityCustomAction) -> Bool`

Discussion

Custom Actions are navigated and selected through methods provided by Assistive Technologies that any user can perform, in contrast with having them mixed with content and selected through gestures users may not be able to perform (see *Shortcuts* in `UIAccessibilityAction` section).

Accessibility Accommodations

These are user-controlled system-wide flags, applications can use them to better accommodate for the user's accessibility requirements surrounding sensory information such as vision (e.g. Bold Text) or sound (e.g. Mono Audio).

These flags might be changed post-application launch, in which case a notification is broadcast by iOS (see: `UIAccessibilityNotifications`)

`.DarkerSystemColorsEnabled`

Returns the status for "Darker Colors".

Declaration

```
func UIAccessibilityDarkerSystemColorsEnabled() -> Bool
```

Discussion

Returns `true` when "Darker Colors" is enabled. Darker Colors is a feature intended to accommodate for brightness sensitivity by using a darker color variant. This feature is handled automatically when using iOS System Colors.

`.IsBoldTextEnabled`

Returns the status for "Bold Text".

Declaration

```
func UIAccessibilityIsBoldTextEnabled() -> Bool
```

Discussion

Returns `true` when "Bold Text" is enabled. Bold Text is a feature intended to provide better contrast and readability by using a bold font variant in place of regular or lighter ones. This feature is handled automatically if using iOS System Fonts (i.e. Helvetica Neue, San Francisco) and Auto-Layout. Custom fonts and manual layout require additional work to provide the correct variant and readjust layout elements.

`.IsClosedCaptioningEnabled`

Returns the status for "Closed Captioning + SDH".

Declaration

```
func UIAccessibilityIsClosedCaptioningEnabled() -> Bool
```

Discussion

Returns `true` when “CC + SDH“ is enabled. Closed Captioning + SDH (Subtitles for the Deaf and Hard of Hearing) is a feature intended to make information contained in visual or audio media accessible to anyone be conveying it through text, allowing for automatic translation or alternative forms of perception such as braille. SDH subtitles include not only dialog (e.g. “Hello”) but also other sounds such as folly (e.g. “door creeks”) and scene descriptions (e.g “a woman sits by the window”). This is handled automatically when using AVFoundation and a CC + SDH track is available.

`.IsGrayscaleEnabled`

Returns the status for “Grayscale”.

Declaration

```
func UIAccessibilityIsGrayscaleEnabled() -> Bool
```

Discussion

Returns `true` when “Grayscale” is enabled. Grayscale is a feature intended to help colorblind users distinguish color by shade. Desaturation occurs at a system-level, this flag serves only to allow for adjustments, if necessary in order to avoid loosing information or functionality.

Examples

Refer to accessibility guideline “1.4.1 Use of Color (Level A)” for examples.

`.IsGuidedAccessEnabled`

Returns the status for “Guided Access”.

Declaration

```
func UIAccessibilityIsGuidedAccessEnabled() -> Bool
```

Discussion

Returns `true` when “Guided Access” is enabled. Guided Access is a feature intended to help users focus on a specific task by imposing certain restrictions such as navigating away from the current app. It’s also possible to know which restrictions are in place (see `UIGuidedAccessRestrictionStateForIdentifier`), apps can use this information to potentially adjust their text copy or features.

Examples

Example #1: Acknowledging the user cannot leave the app and adjusting copy accordingly, replacing the target in prompts such as “Please enable Location Services” to “Ask a supervisor to enable Location Services”.

Example #2: Acknowledging a restriction is in place and handle unavailable features such as opening other apps for directions, web browsing.

`.IsInvertColorsEnabled`

Returns the status for “Invert Colors”

Declaration

```
func UIAccessibilityIsInvertColorsEnabled() -> Bool
```

Discussion

Returns `true` when “Invert Colors” is enabled. Invert Colors is a feature intended to increase contrast for users with low-vision, and reduce screen brightness for color-sensitive users. Colors are inverted at a system-level, this flag serves only to allow for adjustments, if necessary in order to avoid loosing information or functionality.

Examples

Example 1: An app might apply additional processing to images to ensure their information remains correct, such processing might involve inverting images to cancel the system-level invert filter, and subsequently lowering their brightness to respect the user’s preference.

`.IsMonoAudioEnabled`

Returns the status for “Mono Audio”.

Declaration

```
func UIAccessibilityIsMonoAudioEnabled() -> Bool
```

Discussion

Returns `true` when “Mono Audio“ is enabled. Mono Audio is a feature intended to accommodate for users with Unilateral Hearing Loss (UHL) by merging Stereo audio information into a single channel, delivered to the working ear. Audio channel mixing occurs at a system-level, this flag serves only to allow for adjustments, if necessary in order to avoid loosing information or functionality.

`.IsReduceMotionEnabled`

Returns the status for “Reduce Motion”.

Declaration

```
func UIAccessibilityIsReduceMotionEnabled() -> Bool
```

Discussion

Returns `true` when “Reduce Motion” is enabled. Reduce Motion is a feature intended to accommodate for users for whom animations covering a large part of the screen cause discomfort. Motion is reduced at a system-level, with built-in components such as `UIMotionEffect` adapting automatically to this preference. Other methods for implementing animations might require additional work.

`.IsShakeToUndoEnabled`

Returns the status for “Shake to Undo“

Declaration

```
func UIAccessibilityIsShakeToUndoEnabled
```

Discussion

Returns `true` when “Shake to Undo“ is enabled. Shake to Undo is a feature intended to allow users to undo editing by shaking their iOS device. Users with physical and motor impairments

`.IsSpeakScreenEnabled`

Returns the status for “Speak Screen”.

Declaration

```
func UIAccessibilityIsSpeakScreenEnabled() -> Bool
```

Discussion

Returns `true` when “Speak Screen“ is enabled. Speak Screen is a feature intended to help users with low or no vision by speaking the screen contents on demand (e.g. swipe down with to fingers from the top of the screen).

`.IsSpeakSelectionEnabled`

Returns the status for “Speak Selection”.

Declaration

```
func UIAccessibilityIsSpeakSelectionEnabled() -> Bool
```

Discussion

Returns `true` when “Speak Selection“ is enabled. Speak Selection is a feature intended to help users with low or no vision by speaking the selected screen contents on demand (e.g. highlighting a block of text and selecting “Speak” from the tooltip menu).

`.IsSwitchControlRunning`

Returns the status for “Switch Control“.

Declaration

```
func UIAccessibilityIsSwitchControlRunning() -> Bool
```

Discussion

Returns `true` when “Switch Control“ is enabled. Switch Control is a feature intended to help users with mobility restrictions by allowing all interaction to be done via a single switch which can be triggered by any means possible (including facial movements).

`.IsVoiceOverRunning`

Returns the status for “VoiceOver”.

Declaration

```
func UIAccessibilityIsVoiceOverRunning() -> Bool
```

Discussion

Returns `true` when “VoiceOver” is enabled. VoiceOver is a feature intended to help users with low or no vision by converting information into a format that’s most accessible to the user (e.g. speech or braille).

Examples

Example #1: When VoiceOver is running, elements that usually disappear quickly from screen, used to convey events such as content refresh, can be made to persist for longer, and their contents conveyed to the user via a `UIAccessibilityAnnouncement` (e.g. “Email refreshed, 4 new items”).

Accessibility Notifications

An `NSNotificationCenter`-based API is available for observing and posting Accessibility-related notifications.

Topics include general Assistive Technology state and control, Accessibility Accommodations state and Content changes.

Among other things, this API might be used to inform the user of any arbitrary event such as content refresh, avoid interrupting an ongoing announcement, adjust the app to preferences such as Bold Text, request for certain Assistive Technologies to be enabled, or direct Accessibility Zoom to a new element.

Assistive Technology Identifiers

System-defined variables for identifying Assistive Technologies. Ensures code validity across operating system releases, correct operation of API methods for pausing and resuming Assistive Technologies (i.e. `.PauseAssistiveTechnologyNotification`, `.ResumeAssistiveTechnologyNotification`)

`.SwitchControlIdentifier`

Identifier for Switch Control.

Declaration

```
let UIAccessibilityNotificationSwitchControlIdentifier: String
```

`.VoiceOverIdentifier`

Identifier for VoiceOver.

Declaration

```
let UIAccessibilityNotificationVoiceOverIdentifier: String
```

Post

Accessibility Notifications are broadcast using `UIAccessibilityPostNotification` function.

Declaration

```
public func UIAccessibilityPostNotification(notification: UIAccessib\
ilityNotifications, _ argument: AnyObject?)_
```

`.AnnouncementNotification`

Post to convey an announcement to Assistive Technology users.

Declaration

```
var UIAccessibilityAnnouncementNotification: UIAccessibilityNotifica\
tions
```

Discussion

Posted using `UIAccessibilityPostNotification`, argument should be a String to be announced by Assistive Technologies (e.g. converted by VoiceOver to speech or braille). Commonly used to convey information that does not update the user interface (UI), or updates the UI briefly.

Examples

```
UIAccessibilityPostNotification(UIAccessibilityAnnouncementNotificat\
ion, "Email refreshed, 4 New Emails")
```

`.PauseAssistiveTechnologyNotification`

Post to pause a specific Assistive Technology.

Declaration

```
var UIAccessibilityPauseAssistiveTechnologyNotification: UIAccessibi\
lityNotifications
```

Discussion

Posted using `UIAccessibilityPostNotification`, argument is a Assistive Technology Identifier. Temporarily pauses the specified Assistive Technology, until `UIAccessibilityResumeAssistiveTechnologyNotification` is called with the same argument, or the user performs an action that requires the Assistive Technology to resume operations.

Examples

Example #1: An application may pause Switch Control until an animation is completed, as contents may shift making selecting the desired element harder.

`.ResumeAssistiveTechnologyNotification`

Post to resume a specific Assistive Technology.

Declaration

```
var UIAccessibilityResumeAssistiveTechnologyNotification: UIAccessib\
ilityNotifications
```

Discussion

Posted using `UIAccessibilityPostNotification`, argument is a Assistive Technology Identifier.

`.RegisterGestureConflictWithZoom`

Used to inform and resolve a conflict between application-specific and system-defined accessibility Zoom gestures.

Declaration

```
func UIAccessibilityRegisterGestureConflictWithZoom()
```

Discussion

The system-defined accessibility Zoom gestures are a set of 3-finger gestures used to toggle, control magnification, and target. Applications that register 3-Finger gestures should offer alternative ways to access the same functionality, as well as allow users to disable application-specific 3-finger gestures and continue using accessibility Zoom.

Examples

Example #1: An application that uses 3-Finger gestures to control the edit history, offering a 3-Finger horizontal drag to Undo or Redo, allows the same functionality to be performed via Undo and Redo buttons, and allows Assistive Technology users to disable the application-specific gestures through `UIAccessibilityRegisterGestureConflictWithZoom`.

`.RequestGuidedAccessSession`

Programmatically enable Guided Access.

Declaration

```
func UIAccessibilityRequestGuidedAccessSession(_ enable: Bool, _ com\
pletionHandler: (Bool) -> Void)
```

Discussion

Set `enable` to `true` to enable Guided Access, `false` for the opposite effect. The request will only succeed on Supervised devices where the bundle identifier for the requesting app has been whitelisted for on-demand Guided Access. It is the app's responsibility to balance this request and disable Guided Access.

Examples

Example #1: An app whitelisted for Guided Access on a Supervised device, such as a Kiosk app in a public space, may enable and run under Guided Access to prevent visitors from unintended usage of the device.

`.ZoomFocusChanged`

Inform Assistive Technologies to switch Accessibility Zoom focus to a specific element.

Declaration

```
func UIAccessibilityZoomFocusChanged(_ type: UIAccessibilityZoomType\
, _ frame: CGRect, _ view: UIView)


public enum UIAccessibilityZoomType : Int {
    case InsertionPoint
}
```

Discussion

Specify the `UIAccessibilityZoomType`, currently only available to specify changes in text insertion points,, along with the frame of the focused view, and the focused view itself. As of iOS 9.3 there is no way to determine enabled status for Accessibility Zoom, while no work is needed for built-in components, focus change on custom input fields should be flagged using this method.

Examples

Example #1: An application that prompts for login credentials informs ATs using this method when a user moves between form fields.

Observe

Accessibility Notification observers should be managed using the standard `NSNotificationCenter` methods.

Assistive Technology State Changes

Notifications for Assistive Technologies status changes.

`.AnnouncementDidFinishNotification`

Posted when the Assistive Technologies have finished reading an announcement.

Declaration

```
let UIAccessibilityAnnouncementDidFinishNotification: String
```

Notification userInfo dictionary:

```
[
]()// The announced string
UIAccessibilityAnnouncementKeyStringValue: String,
    // The announcement result, to be interpreted as NSNumber.boolVa\
lue
UIAccessibilityAnnouncementKeyWasSuccessful: NSNumber
]
```

Discussion

Assistive Technologies read out announcements in sequence to avoid overwhelming the user. For this reason applications are required to exert moderation when posting notifications, using `AnnouncementDidFinishNotification` as a semaphore, or have them discarded.

`.GuidedAccessStatusDidChangeNotification`

Posted when Guided Access is toggled.

Declaration

```
let UIAccessibilityGuidedAccessStatusDidChangeNotification: String
```

Discussion

Use `UIAccessibilityIsGuidedAccessRunning` for current status, this notification does not include a parameter.

`.SpeakScreenStatusDidChangeNotification`

Posted when Speak Screen is toggled.

Declaration

```
let UIAccessibilitySpeakScreenStatusDidChangeNotification: String
```

Discussion

Use `UIAccessibilityIsSpeakScreenEnabled` for current status, this notification does not include a parameter.

`.SpeakSelectionStatusDidChangeNotification`

Posted when Speak Selection is toggled.

Declaration

```
let UIAccessibilitySpeakSelectionStatusDidChangeNotification: String
```

Discussion

Use `UIAccessibilityIsSpeakSelectionEnabled` for current status, this notification does not include a parameter.

`.SwitchControlStatusDidChangeNotification`

Posted when Switch Control is toggled.

Declaration

```
let UIAccessibilitySwitchControlStatusDidChangeNotification: String
```

Discussion

Use `UIAccessibilityIsSwitchControlRunning` for current status, this notification does not include a parameter.

`.VoiceOverStatusChanged`

Posted when VoiceOver is toggled.

Declaration

```
let UIAccessibilityVoiceOverStatusChanged: String
```

Discussion

Use `UIAccessibilityIsVoiceOverRunning` for current status, this notification does not include a parameter.

Accessibility Accommodations State Changes

Notifications for Accessibility Accommodations status changes.

`.BoldTextStatusDidChangeNotification`

Posted when Bold Text is toggled.

Declaration

```
let UIAccessibilityBoldTextStatusDidChangeNotification: String
```

Discussion

Use `UIAccessibilityIsBoldTextEnabled` for current status, this notification does not include a parameter.

`.InvertColorsStatusDidChangeNotification`

Posted when Invert Colors is toggled.

Declaration

```
let UIAccessibilityInvertColorsStatusDidChangeNotification: String
```

Discussion

Use `UIAccessibilityIsInvertColorsEnabled` for current status, this notification does not include a parameter.

`.DarkerSystemColorsStatusDidChangeNotification`

Posted when Darker System Colors is toggled.

Declaration

```
let UIAccessibilityDarkerSystemColorsStatusDidChangeNotification: St\
ring
```

Discussion

Use `UIAccessibilityDarkerSystemColorsEnabled` for current status, this notification does not include a parameter.

`.GrayscaleStatusDidChangeNotification`

Posted when Grayscale is toggled.

Declaration

```
let UIAccessibilityGrayscaleStatusDidChangeNotification: String
```

Discussion

Use `UIAccessibilityIsGrayscaleEnabled` for current status, this notification does not include a parameter.

`.MonoAudioStatusDidChangeNotification`

Posted when Mono Audio is toggled.

Declaration

```
let UIAccessibilityMonoAudioStatusDidChangeNotification: String
```

Discussion

Use `UIAccessibilityIsMonoAudioEnabled` for current status, this notification does not include a parameter.

`.ClosedCaptioningStatusDidChangeNotification`

Posted when Closed Captioning is toggled.

Declaration

```
let UIAccessibilityClosedCaptioningStatusDidChangeNotification: Stri\
ng
```

Discussion

Use `UIAccessibilityIsClosedCaptioningEnabled` for current status, this notification does not include a parameter.

`.ReduceMotionStatusDidChangeNotification`

Posted when Reduce Motion is toggled.

Declaration

```
let UIAccessibilityReduceMotionStatusDidChangeNotification: String
```

Discussion

Use `UIAccessibilityIsReduceMotionEnabled` for current status, this notification does not include a parameter.

`.ReduceTransparencyStatusDidChangeNotification`

Posted when Reduce Transparency is toggled.

Declaration

```
let UIAccessibilityReduceTransparencyStatusDidChangeNotification: St\
ring
```

Discussion

Use `UIAccessibilityIsReduceTransparencyEnabled` for current status, this notification does not include a parameter.

`UIContentSizeCategoryDidChangeNotification`

Posted when the preferred Content Size Category changes.

Declaration

```
let UIContentSizeCategoryDidChangeNotification: String
```

Notification userInfo dictionary:

```
[
]()// The new Content Size Category
UIContentSizeCategoryNewValueKey: String,
]
```

The Content Size Category can also be read from `UIApplication`’s `preferredContentSizeCategory` property.

```
var preferredContentSizeCategory: String { get }
```

Base Content Sizes

Base Content Size Categories range between base XS and XXXL values, defaulting to `UIContentSizeCategoryLarge`.

- `UIContentSizeCategoryExtraSmall: String`
- `UIContentSizeCategorySmall: String`
- `UIContentSizeCategoryMedium: String`
- `UIContentSizeCategoryLarge: String`
- `UIContentSizeCategoryExtraLarge: String`
- `UIContentSizeCategoryExtraExtraLarge: String`
- `UIContentSizeCategoryExtraExtraExtraLarge: String`

Large Accessibility Content Sizes

Users can enable “Large Accessibility Content Sizes”, adding 5 sizes larger than the base `UIContentSizeCategoryExtraExtraExtraLarge`.

- `UIContentSizeCategoryAccessibilityMedium: String`
- `UIContentSizeCategoryAccessibilityLarge: String`
- `UIContentSizeCategoryAccessibilityExtraLarge: String`
- `UIContentSizeCategoryAccessibilityExtraExtraLarge: String`
- `UIContentSizeCategoryAccessibilityExtraExtraExtraLarge: String`

Discussion

To better accommodate for varying levels of visual acuity, apps supporting Dynamic Type and Auto Layout can use a smaller or larger font size to display content.

For built-in fonts, pre-calculated adjustments exist for each of the available content size categories.

These adjustments are encapsulated in `UIFontDescriptors`, and the preferred size category, set by users via the “Large Text” screen within Accessibility Settings, is used to determine the appropriate `UIFontDescriptor`.

Properties within a given `UIFontDescriptor` will then dictate the size of the text and the intrinsic content size of the element in which that text is used.

As such, built-in fonts and components, require no work to display a font size that matches user’s preferred content size, requiring only a call to `invalidateIntrinsic-ContentSize()` when a `UIContentSizeCategoryDidChangeNotification` is broadcast.

Custom fonts and components may require the calculator of appropriate UIFontDescriptors to match all possible content sizes, and manual layout on initialization and content size category change.

The preferred content size is available in the userInfo dictionary broadcast with this notification, as well as on-demand in `UIApplication.sharedApplication().preferredContentSize`

Content Changes

Notifications for application content changes.

`.LayoutChangedNotification`

Posted when the layout of a screen changes.

Declaration

```
var UIAccessibilityLayoutChangedNotification: UIAccessibilityNotific\
ations
```

Discussion

Posted using `UIAccessibilityPostNotification`, argument is either a String or the element that should be focused by Assistive Technologies. Commonly used to convey changes due to the addition or removal of on-screen elements.

Examples

```
    // I.e. “Panel hidden”
UIAccessibilityPostNotification(UIAccessibilityLayoutChangedNotifica\
tion, String)


    // I.e. self.panelViewIBOutlet
UIAccessibilityPostNotification(UIAccessibilityLayoutChangedNotifica\
tion, UIAccessibilityElement)
```

`.ScreenChangedNotification`

Posted when a major portion of the screen changes.

Declaration

```
var UIAccessibilityScreenChangedNotification: UIAccessibilityNotific\
ations
```

Discussion

Posted using `UIAccessibilityPostNotification`, argument is either a String or the element that should be focused by Assistive Technologies. Commonly used to convey changes to large portions of the screen, due to the addition or removal of on-screen elements.

Examples

```
    // I.e. “Major view hidden”
UIAccessibilityPostNotification(UIAccessibilityScreenChangedNotifica\
tion, String)
```

```
    // I.e. self.majorViewIBOutlet
UIAccessibilityPostNotification(UIAccessibilityLayoutChangedNotifica\
tion, UIAccessibilityElement)
```

`.PageScrolledNotification`

Posted after the `accessibilityScroll` method has been called.

Declaration

```
var UIAccessibilityPageScrolledNotification: UIAccessibilityNotifica\
tions
```

Discussion

Posted using `UIAccessibilityPostNotification`, argument should be a String to be announced by Assistive Technologies (e.g. converted by VoiceOver to speech or braille). Commonly used to convey the current screen section to the user (e.g. Tabs or Pages).

Assistive Technologies will inform the user has reached a boundary when the same string is repeated.

Examples

```
UIAccessibilityPostNotification(UIAccessibilityPageScrolledNotificat\
ion, "Tab 2 of 4")
```

```
UIAccessibilityPostNotification(UIAccessibilityPageScrolledNotificat\
ion, "Page 42 of 100")
```

Control Events

Events for interactive objects.

Declaration

```
struct UIControlEvents : OptionSetType
```

`.AllEvents`

When any event occurs, including system events.

Declaration

```
var AllEvents: UIControlEvents { get }
```

`.AllTouchEvents`

Catch-all for all touch events.

Declaration

```
var AllTouchEvents: UIControlEvents { get }
```

Discussion

Posted when any `UIControlEvent` related to touch is posted (e.g. `UIControlEvent-TouchDown`).

`.AllEditingEvents`

Catch-all for all editing events.

Declaration

```
var AllEditingEvents: UIControlEvents { get }
```

Discussion

Posted when any `UIControlEvent` related to editing is posted (e.g. `UIControlEventE-ditingDidBegin`).

`.EditingDidBegin`

When a text input receives focus and content editing begins.

Declaration

```
var EditingDidBegin: UIControlEvents { get }
```

`.EditingDidEnd`

When a touch occurs outside the text input element, causing it to loose focus and ending its editing state.

Declaration

```
var EditingDidEnd: UIControlEvents { get }
```

Discussion

Differs from `EditingDidEndOnExit` as editing ends implicitly by tapping any on-screen element, instead of an explicit tap on a button such as Done or Search.

`.EditingDidEndOnExit`

When a touch occurs on an element dedicated to ending text editing, while the text editing element is still focused.

Declaration

```
var EditingDidEndOnExit: UIControlEvents { get }
```

Discussion

Differs from `EditingDidEnd` as editing ends explicitly by touching a button such as Done or Search, instead of implicitly through a touch outside the text input element.

`.PrimaryActionTriggered`

When the primary action of an element is triggered.

Declaration

```
var PrimaryActionTriggered: UIControlEvents { get }
```

Discussion

Certain OSs such as tvOS do not support touch events and as such the activation of interactive elements will post a `PrimaryActionTriggered` event instead of a `TouchUpInside`. The `PrimaryActionTriggered` event is also supported by iOS for an input-agnostic, cross-OS method for conveying interactive element activation.

Examples

Example #1: An application targeting tvOS and iOS, listens to `PrimaryActionTriggered` to enable code reuse across platforms.

`.TouchDown`

When an element is touched once but not moved or released.

Declaration

```
var TouchDown: UIControlEvents { get }
```

`.TouchDownRepeat`

When an element is touched repeatedly more than once.

Declaration

```
var TouchDownRepeat: UIControlEvents { get }
```

Discussion

Fired for double, triple or more taps in a short amount of time, number of taps exposed through `UITouch.tapCount: Int { get }`.

`.TouchDragInside`

When an element is touched once, and the finger moved within the element bounds without being released.

Declaration

```
var TouchDragInside: UIControlEvents { get }
```

`.TouchDragEnter`

When a finger is dragged into the bounds of the control.

Declaration

```
var TouchDragEnter: UIControlEvents { get }
```

`.TouchDragExit`

When an element is touched once, and the finger leaves the bounds of the element bounds without being released.

Declaration

```
var TouchDragExit: UIControlEvents { get }
```

`.TouchUpInside`

When an element is touched once, and the finger is released within the bounds of the element.

Declaration

```
var TouchUpInside: UIControlEvents { get }
```

Discussion

This corresponds to the common tapping of an element.

`.TouchUpOutside`

When an element is touched once, and the finger is released outside the bounds of the element.

Declaration

```
var TouchUpOutside: UIControlEvents { get }
```

`.TouchCancel`

When an element is touched once, and the touch is cancelled.

Declaration

```
var TouchCancel: UIControlEvents { get }
```

Discussion

Touches can be cancelled by dragging the finger far away from the element, in which case a `TouchDragExit` will precede this event, or due to other system events.

`.ValueChanged`

When the value of an element changes.

Declaration

```
var ValueChanged: UIControlEvents { get }
```

Discussion

Elements such as built-in sliders post this event when their value changes, causing Assistive Technologies to announce their `accessibilityValue` property. Custom elements with an `Adjustable` accessibility trait should also post this event.

Examples

Example #1: A custom color picker that fires this event when changed, causing Assistive Technologies to announce it’s `accessibilityValue` containing a textual description of the currently chosen color.

Customizing Speech

To customize text-to-speech output, it is possible to specify how punctuation should be interpreted, how words should be pronounced by specifying what language they belong to, and convey meaning by adjusting pitch.

Refer to Apple's documentation on `NSAttributedString` or how to assign attributes to strings and ranges within strings.

`.Pitch`

Controls the pitch in which text is spoken.

Declaration

```
let UIAccessibilitySpeechAttributePitch: String
```

Discussion

The value is an `NSNumber` containing a floating-point value in the range 0.0 to 2.0. A normal pitch is represented by the value 1.0, lower values indicate a lower pitch and vice-versa.

Examples

Example #1: A children's book may use different pitches or each character, giving them a distinctive personality.

Example #2: An application may use pitch to convey status for individual items, potentially raising the pitch (to a certain extent) for items for which the status is different from normal such as an unread email, or a item in an error state.

`.Punctuation`

Controls wether punctuation should be spoken literally or interpreted.

Declaration

```
let UIAccessibilitySpeechAttributePunctuation: String
```

Discussion

The value of this key is an NSNumber object interpreted as a Boolean value. When the value is YES, all punctuation in the text is spoken. You might use this for code or other text where the punctuation is relevant.

Examples

Example #1: An application displaying computer code where punctuation should be spoken literally to convey the correct information.

Example #2: An application displaying a phone number or any other text where punctuation should be spoken literally, the phone number beginning with “07402” should be read as “zero seven four zero two” instead of “seven-thousand four-hundred and two”.

`.Language`

Helps ensure text is correctly pronounced by specifying its language.

Declaration

```
let UIAccessibilitySpeechAttributeLanguage: String
```

Discussion

The value of this key is an NSString object containing a BCP 47 language code (see below). When applied to text in a string, the rules for the specified language govern how that string is pronounced.

The BCP 47 language code is a combination of an three/two-letter ISO 639-1 language code and a two-letter ISO 3166-1 region code (e.g. en-US, cpp-ST).

Helpers

Functions to help implementing accessibility. Includes convention of view frames and paths to absolute screen coordinates, as well as determining guided access restrictions.

`UIAccessibilityConvertFrameToScreenCoordinates`

Converts UIView frame coordinates relative to the view's hierarchy, to absolute screen coordinates.

Declaration

```
func UIAccessibilityConvertFrameToScreenCoordinates(_ rect: CGRect, \
_ view: UIView) -> CGRect
```

Discussion

Can be used to adjust the coordinates passed to `accessibilityFrame`.

Examples

Example #1: A view A1 with the frame coordinates `0, 0, view.width, view.height` embedded in a view A with the frame coordinates `100, 100, view.width, view.height` will have the absolute screen coordinates of `100, 100, view.width, view.height`.

`UIAccessibilityConvertPathToScreenCoordinates`

Converts UIView path coordinates relative to the view's hierarchy, to absolute screen coordinates.

Declaration

```
func UIAccessibilityConvertPathToScreenCoordinates(_ path: UIBezierP\
ath, _ view: UIView) -> UIBezierPath
```

Discussion

Can be used to adjust the coordinates passed to `accessibilityPath`.

Examples

Example #1: A view A1 with the path coordinates `0, 0, view.width, view.height` embedded in a view A with the coordinates `100, 100, view.width, view.height` will have the absolute screen coordinates of `100, 100, view.width, view.height`.

`UIGuidedAccessRestrictionStateForIdentifier`

Returns the restriction state for the specified Guided Access restriction.

Declaration

```
func UIGuidedAccessRestrictionStateForIdentifier(_ restrictionIdenti\
fier: String) -> UIGuidedAccessRestrictionState
```

`UIGuidedAccessRestrictionState`

The state of a restriction, either allow or deny.

```
UIGuidedAccessRestrictionState : Int {
    case Allow
    case Deny
}
```