Developing Applications for Everyone

Session ???

Tamar E. Granor
Voice: 215-635-1958
Email: tamar_granor@compuserve.com

Overview

For applications to be truly usable for everyone, including individuals with physical challenges such as visual impairments, hearing difficulties, and problems with mobility, they must be designed for accessibility from day one. The good news is that the process of ensuring that people with disabilities can use an application is likely to improve the interface for all users. This session will look at the design issues involved, tools available to make applications accessible to all users, and demonstrate techniques in VFP that improve the user interface as well as increasing accessibility.

Why worry about physical challenges?

When you hear that someone is "physically challenged," "disabled" or "handicapped," you probably think of a person in a wheelchair or a blind person. However, many people have smaller disabilities that impact their use of computers. Overall, about 20% of Americans have one or more physical disabilities. (Most of the statistics in this section apply to Americans, solely because those statistics are more readily available than numbers for other countries or the whole world.)

One of the most common disabilities is reduced vision. As people age, their vision deteriorates, both in acuity and color perception. By their mid-40's, many people notice that they can't read what they could before. According to one source (AllAboutVision.Com), 17% of Americans 45 and over report "some type of vision impairment even when wearing glasses or contacts."

There are other kinds of visual disabilities besides total blindness or low vision, too. Among them, color blindness is extremely common. Somewhere between 5% and 10% of men and about 1% of women have some form of color blindness.

Hearing impairments (either deafness or reduced hearing) affect more than 8% of Americans. Again, this disability affects people more as they age. By age 65, about a third of Americans have some hearing loss.

Mobility impairments cover a wide range of issues, including paralysis, muscle weakness, poor muscle control, joint movement limitations, and missing limbs or digits. Over a million Americans use a wheelchair, while about 6.5 million use some form of mobility aid.

Finally, seizure disorders affect about 1 person in 15. For some, seizures are chronic while others have only a single seizure and never have a recurrence.

These statistics offer the strongest reason for ensuring that those with disabilities can use your applications. With one-fifth of the population coping with at least one disability, creating applications that exclude users with disabilities is likely to have serious economic consequences.

However, there are other reasons as well. The best known is the legal reason. In the US, the Americans with Disabilities Act, passed in 1990, guarantees a "reasonable accommodation" in both work and living to people with disabilities. Companies with 15 or more employees are affected by these regulations, and their employees can reasonably expect them to provide accessible software and hardware. Other countries have similar laws. For example, the Disability Discrimination Act of 1995 sets out the rights of people with disabilities in the UK.

There's a financial incentive as well. Section 508 of the Rehabilitation Act enforces guidelines like those of the ADA on the federal government and any organization receiving federal funds. Since the federal government is the largest purchaser of software in the US, providing accessible software could make the difference between success and failure in the marketplace.

Finally, there's an ethical reason to make software accessible to the widest possible range of people. Put simply, it's the right thing to do.

What are the issues?

Many applications are difficult or impossible for those with physical challenges to use. The problems vary depending on the individual user's disabilities.

Vision-related problems

For users who are color-blind, there's one basic problem. Application developers often hard-code color choices. If those colors don't provide sufficient contrast, this user may not be able to distinguish the various components of the application.

Color contrast is also an issue for users with reduced vision. For these users, however, the size of text is also an issue. Small lettering may be unreadable for them. Users with impaired vision may also find it difficult to read text that appears on a patterned background.

Users who are totally blind generally use a screen reader program to tell them what's being displayed. These people can't use any software that doesn't cooperate with such tools.

Hearing-related problems

In general, most applications present fewer problems for people with hearing disabilities than for those with other disabilities. However, applications that provide instructions using sounds or other audio with no visual indicators do present an obstacle to users who are totally deaf or hearing-impaired. Sound-only instructions can also be a problem for applications that run in a noisy environment, like a factory floor.

Mobility-related problems

There are a variety of accessibility issues for people with mobility impairments. Some users are unable to use a mouse, so need all options to be available from the keyboard. Others can use a mouse, but can't manage fine control of it. For other users, the keyboard provides impediments. For example, some users are unable to press key combinations. Other users need extra time to make choices or enter data. Some users are unable to use their hands at all and must use a mouth stick, eye movement or voice for input.

How can applications be accessible?

Making applications accessible takes in a variety of approaches. Some solutions are hardware-based, such as providing alternative input devices. However, even when such devices are used, the software must be able to receive input from them.

Other solutions are system-based. Windows includes a number of tools that provide access for users with disabilities. The set of tools available has grown in recent versions of Windows. In addition, more of the tools are built right into newer Windows versions, while older ones required add-ons.

Finally, individual applications make themselves accessible in several ways. The first is by supporting the accessibility tools. In addition, applications can help users with disabilities by following Windows standards and by applying the user's Windows choices in such things as colors and sounds.

What Accessibility tools does Windows provide?

Recent versions of Windows offer a large collection of tools that aid users with disabilities. The complete list of built-in tools is available at http://www.microsoft.com/enable/products/chartwindows.htm. Here's an overview of Windows' accessibility features. In the discussions below, "applications" generally means both Windows and application programs running in Windows.

SoundSentry

This tool tells applications to provide a visual indication that a sound has been played. The user can choose whether the indicator is a flash of the active window's title bar, a flash of the entire active window or a flash of the desktop.

ShowSounds

This tool goes farther than SoundSentry. Think of it as closed captioning for applications. It tells applications to show text on screen when speech or a sound is played. Unfortunately, in my testing, I couldn't find any applications that supported this capability.

ToggleKeys

When enabled, this tool plays a sound when any of the keyboard toggle keys (Caps Lock, Num Lock or Scroll Lock) is hit. A different sound is played when the toggle is turned on than when it's turned off.

StickyKeys

For some users, holding down a key combination is extremely difficult or impossible. Turning StickyKeys on indicates that the modifier keys (Alt and Ctrl) can be used in sequence with other keys rather than simultaneously.

FilterKeys

Some users with impaired motor skills tend to hit keys accidentally or to hold keys down too long (so they start to repeat). This feature lets the user customize the behavior of the keyboard in several ways. The user can indicate that repeated keystrokes are to be ignored or can set a time threshold that must occur between keystrokes to allow the same key to be repeated. The user can also indicated how long a key must be pressed to be accepted and how long a key must be pressed to begin repeating.

MouseKeys

This setting lets users perform mouse functions from the keyboard. It turns the numeric keypad into a mouse substitute, using most of the numeric keys to navigate, the 5 key for click and the +

key for double-click. Keys are also designated to indicate to which button a click or double-click applies.

SerialKeys

This feature enables the use of alternative input devices attached to the computer's serial port. The device can replace the keyboard or the mouse and must be programmed to send the appropriate key definitions. This Knowledge Base article provides detailed information on setting up an alternative input device: http://support.microsoft.com/support/kb/articles/Q260/5/17.ASP.

Sound schemes

Windows allows users to choose the sounds played for various system actions. Individual sounds can be organized into schemes. Users with impaired vision can choose sound schemes or design their own to provide unique sounds for a variety of system and application actions. Users with hearing impairments can choose sounds that they are able to hear.

Color schemes

As with sounds, users can customize the colors they see in Windows. Users who are color-blind or have reduced vision can choose colors that maximize visibility for them. The built-in high-contrast color schemes are specifically designed to aid users with visual disabilities. These color schemes are available in a variety of sizes as well.

Pointer schemes

The icons used to represent the mouse pointer can also be customized. The schemes provided include some with larger icons and, in some versions of Windows, inverted colors to improve contrast.

Pointer control

The different versions of Windows provide various options for controlling the mouse (or other pointer). Among the items that can be set by the user are the speed at which the pointer moves and the speed at which it accelerates as it continues to move, the speed of a double-click, and the actions assigned to the different mouse buttons (that is, which button corresponds to the primary click and which means "right-click").

Users can also determine whether the mouse immediately moves to the default button in dialogs, and, in some versions, whether the pointer disappears while the user is typing.

The ClickLock feature lets the user set the mouse to highlight or drag without having to hold the button down. The length of time the button must be held down to begin highlighting or dragging is configurable. This feature is available with all pointer devices in newer versions of Windows and only with Intellipoint devices in older versions.

Wheel control

For pointers with a wheel, there's one more configurable feature. The user can determine how much the display scrolls when the wheel is rolled.

Magnifier

This tool provides an on-screen magnifying glass. A dockable window (docked at the top by default) displays a blown-up version of the display. The magnification factor is configurable, as are several other behaviors. Magnifier is built into newer versions of Windows.

Narrator

This tool reads Windows aloud. Aimed at users with serious visual handicaps, Narrator uses a computer-synthesized voice to identify the active window and to read windows contents. Narrator has several options, including modifying the synthesized voice. Narrator is included only in Windows 2000 and Windows XP.

On-Screen Keyboard

For users with impaired mobility who cannot use a keyboard, the On-Screen Keyboard provides an alternate method of typing. The tool displays a keyboard on the screen and allows the user to choose characters by clicking or by holding the mouse over the character (the time necessary to consider a key chosen is configurable). A third choice scans the "keyboard" continuously and makes a choice based on a single input from the user. First, rows are highlighted and user input selects a row. Then, the keys on that row are highlighted one by one and an input chooses that key. The keyboard itself can be configured in several ways, including the number of keys. The On-Screen Keyboard is provided with Windows 2000 and later.

Other tools

Many third-party accessibility tools are available and, in fact, Microsoft indicates that Magnifier, Narrator and the On-Screen Keyboard aren't really meant for full-time use. For a list of third-party accessibility tools, go to http://www.microsoft.com/enable/products/aids.asp.

Voice input

Although Windows doesn't provide any tools for voice input, it does support it and there are a number of speech recognition products that allow users to work within Windows. Such tools are valuable for users with mobility impairments that prevent them from using a keyboard or a mouse.

Accessibility settings management

In addition to the actual tools, Windows gives the user several options for managing the accessibility settings. Accessibility settings are stored for individual users of a machine. When saving settings from the Accessibility applet, users can indicate that the current settings should be the default for new users. In Windows 2000 and later, accessibility settings can be stored to a file and retrieved later.

Accessibility settings can be configured to turn off automatically after the system is idle for a specified length of time.

Windows 2000 offers a Utility Manager, which centralizes control of Narrator, Magnifier and On-Screen Keyboard.

What should applications do?

The accessibility tools work at the operating system level, so what role do individual applications play? The role varies depending which accessibility tool you look at. The tools can be divided into two groups – those that do something at the system level (Microsoft calls these "Built-in accessibility features") and those that let the user configure behavior ("Accessibility parameters").

Built-in accessibility features include such things as FilterKeys, MouseKeys and SoundSentry. They're managed at the system level and shouldn't require any special actions by applications.

Accessibility parameters include items like sound schemes, color schemes and pointer schemes, and ShowSounds. Windows provides mechanisms for determining the user's settings and applications should use those settings appropriately.

General Design Issues

There are a number of design choices that help to make an application more accessible. Many of them fall into the general category of adhering to standards.

Using system tools wherever possible makes it easier for accessibility aids to render the application (whether by magnifying it or by reading it to the user). This means both using system dialogs where appropriate and putting text on the screen through the normal text-drawing mechanisms. Among other things, providing text as bitmaps creates difficulties for accessibility aids. Using the system's pointers is another way to assist accessibility aids; typically, they use the system pointer to determine focus. Following the user's color choices also falls into this category.

Consistent design, where each form in an application follows the same pattern, aids users with limited vision and limited mobility as they can learn the pattern and take advantage of it. Not only should forms be consistent within the application, but they should follow operating system guidelines where they exist. Make sure that the tab order on each form follows the visual order in a logical way. Keep in mind that someone using a magnifier can see only a small portion of the screen at once, so consistency becomes far more important than for other users.

Providing keyboard alternatives for all mouse actions enables those who cannot use a mouse to work with the application. (Including all commands in the menu also helps all users learn what options the application provides.) In addition, providing multiple ways to accomplish tasks increases the chance that a given user can find a technique that works for him.

Don't count on pop-up text (like tooltips) to let the user know what his options are. Many accessibility aids can't read tooltips and similar text. Along the same lines, stay away from fake buttons and hot spots since screen readers may not be able to identify them. If you do use one control to mimic another, give it a name that identifies its use rather than its origins.

Make important sounds visible. When sound is used as an adjunct to display, it's not necessary to take special actions, but when sound is used to alert users or provide narration, it's important to provide a non-auditory alternative. The easiest way to accomplish this is to cooperate with existing aids (SoundSentry and ShowSounds). While sound-only techniques can be a problem for users with impaired hearing, keep in mind that adding sounds for feedback can be helpful to users with visual impairments.

Avoid flickering or flashing screens. They can trigger seizures in some people with epilepsy.

For an application to be truly accessible, its documentation must also be accessible. Where possible, make documentation available in a variety of formats, including electronic, print (a variety of print sizes), and audio. When a choice must be made as to which to provide, choose electronic since it can be transformed to the other formats most easily. Avoid making graphics in the documentation essential to comprehension – provide enough description that a user who can't see them can still follow the discussion. Provide Help using standard Help tools, which are open to accessibility aids.

When referring to users with disabilities, whether in the user interface or the documentation, use "people-first" language. That is, talk about "users with disabilities" rather than "disabled users."

What does VFP provide?

Visual FoxPro is a fairly high-level language. As a result, it's possible to use many of the user's settings without having to probe the operating system for them. For others, though, you'll need to make API calls or query the Registry.

Setting colors

One of the easiest things to do in VFP is apply the user's color settings. This is controlled by the ColorSource property of forms and controls. By default, newly created forms pick up the current Windows settings for dialogs; in this case, ColorSource = 4-Windows Control Panel (3D Colors). Change ColorSource to 5-Windows Control Panel (Windows Colors) to use the current settings for documents. For controls, the choices for ColorSource are a little different, but the right choice for creating an accessible application is 4-Windows Colors.

There are times when you need to set the color of a particular object explicitly, generally to make it stand out. In those situations, rather than hard-coding the color, it's best to use the GetSysColor API function to extract an appropriate color from the user's choices and apply it. To use GetSysColor, you must first declare it, like this:

```
DECLARE Integer GetSysColor IN Win32API Integer nIndex
```

To use the function, call it like this:

```
nColor = GetSysColor( nItem )
```

nItem is a value that indicates which color you want from the various settings. Table 1 shows a list of the constant values for the various settings. (Be aware that API function names are casesensitive, so you must reference this function as GetSysColor, not GETSYSCOLOR, Getsyscolor, or another variant.)

Table 1 Windows color constants – Use these constants with GetSysColor() to determine the color of a particular interface element. Items marked as N/A in the last column are set automatically based on other choices.

Constant	Value	Meaning	Setting in Appearance dialog
COLOR_SCROLLBAR	0	Scrollbar color	N/A
COLOR_BACKGROUND	1	Color of the background with no wallpaper	Desktop
COLOR_ACTIVECAPTION	2	Caption of active window	Active Title Bar (color setting)
COLOR_INACTIVECAPTION	3	Caption of inactive window	Inactive Title Bar (color setting)
COLOR_MENU	4	Menu	Menu (color setting)
COLOR_WINDOW	5	Windows background	Window (color setting)
COLOR_WINDOWFRAME	6	Window frame	N/A
COLOR_MENUTEXT	7	Text in menus	Menu (font color setting)
COLOR_WINDOWTEXT	8	Text in windows	Window (font color setting) or Message Box (font color setting)
COLOR_CAPTIONTEXT	9	Text in active window caption	Active Title Bar (font color setting)
COLOR_ACTIVEBORDER	10	Border of active window	Active Window Border
COLOR_INACTIVEBORDER	11	Border of inactive window	Inactive Window Border
COLOR_APPWORKSPACE	12	Background of MDI desktop	Application Background
COLOR_HIGHLIGHT	13	Selected item background	Selected Items (color setting)
COLOR_HIGHLIGHTTEXT	14	Selected menu item (text color)	Selected Items (font color setting)

COLOR_BTNFACE	15	Button	3D Objects (color setting)
COLOR_BTNSHADOW	16	3D shading of button	N/A
COLOR_GRAYTEXT	17	Disabled text	N/A
COLOR_BTNTEXT	18	Button text	3D Objects (font color setting)
COLOR_INACTIVECAPTIONTEXT	19	Text of inactive window caption	Inactive Title Bar (font color setting)
COLOR_BTNHIGHLIGHT	20	3D highlight of button	N/A
COLOR_3DDKSHADOW	21	Edge color for dark side of 3-D objects	N/A
COLOR_3DLIGHT	22	Edge color for light side of 3-D objects	3D Objects (color setting)
COLOR_INFOTEXT	23	Tooltip text color	Tooltip (font color setting)
COLOR_INFOBK	24	Tooltip background color	Tooltip (color setting)
COLOR_HOTLIGHT	26	Color for hot-tracked item (Win 98 and later)	N/A
COLOR_2NDACTIVECAPTION	27	Second active window color for gradient title bars (Win 98 and later)	Active Title Bar (Color 2)
COLOR_2NDINACTIVECAPTION	28	Second inactive window color for gradient title bars (Win 98 and later)	Inactive Title Bar (Color 2)
COLOR_MENUHILIGHT	29	Menu item highlight for flat menus (Win XP only)	N/A
COLOR_MENUBAR	30	Menu background color for flat menus (Win XP only)	N/A

For example, you might choose to use the Selected Item color as the border color for a shape or image. Along the same lines, you might choose to use the Selected Item text color as the fill color for a shape. These colors can make the shape or image stand out while still honoring the user's choices and providing appropriate contrast between the colors used. Here's the code you can use in a Shape object's Init:

```
LOCAL nHighlightColor, nFillColor

DECLARE INTEGER GetSysColor IN Win32API INTEGER nIndex

nHighlightColor = GetSysColor(13)
nFillColor = GetSysColor(14)

This.BorderColor = nHighlightColor
This.FillColor = nFillColor
```

There is one problem with setting colors this way. A user may change his color scheme while the form is running. The form ChangeHighlight.SCX in the session materials demonstrates a technique that keeps the form colors in synch with the user's colors. It uses the ActiveX SysInfo control to respond to changes in the system colors.

Using System Pointers

As with colors, it's easy to apply the user's chosen pointers in VFP. The MousePointer property of forms and controls lets you specify which pointer to use when the mouse is over a particular object. Set it to 0 (the default) to use the normal pointer for that kind of object. Set it to one of the other values to change the pointer type. The appropriate cursor is used, based on what's been specified in the Mouse applet.

The list of choices for MousePointer doesn't include all the pointer types that can be specified – a few are omitted. However, finding out what cursor a user has specified for a given task is quite hard. While pointer information is stored in the Registry under the key HKEY_CURRENT_USER\Control Panel\Cursors, if the user has never specified a cursor other than the default, that key has no values. If the user has changed cursors, but is now using the default, the key has values, but the values have only names and no data. The default pointers don't exist as files – they're Windows resources. Extracting them in a form that VFP can use is a non-trivial task and probably not worth the trouble it would take, given the range of pointers available through MousePointer.

Using System Sounds

Getting your hands on the user's chosen sounds is more difficult. They're stored in the Registry and you have to go fishing to pull them out. Before looking at what's involved, let's see how to play a sound once you access it. The PlaySound API function can play both system sounds and WAV files. You declare it like this:

```
DECLARE INTEGER PlaySound IN winmm.dll
   STRING cName, INTEGER hModule, INTEGER nFlags
```

Once the function is declared, you can call it. For our purposes, you can just pass 0 for the second parameter (hModule). The first parameter is either the filename (with path) of the WAV file to play or the name of a system sound. When you play a WAV file, pass 0 for the nFlags parameter; to play a system sound, pass 0x10000 ("0x" indicates a hexadecimal value). For example, to play the "Asterisk" sound, use:

```
PlaySound( "SystemAsterisk", 0, 0x10000)
```

To play the file Chord.Wav (which is the default for the Asterisk sound), use:

```
PlaySound( "Chord.Wav", 0, 0)
```

The next step is finding the sound you want. Sounds defined in Windows can be played by giving their names, as in the first example above. But, as the example also shows, the internal names of the sounds aren't the same ones displayed in the Sounds applet. To find the internal names of sounds, look in the Registry under the key HKEY_CURRENT_USER\AppEvents\EventLabels. There's a list there of all the system sounds and the associated value for each is the name shown in the Sounds applet.

Another alternative is to define sounds for your application and store them in the Registry. That allows the user to manage them through the Sounds applet, just like any other sounds. The easiest way to add your sound items to the registry and to retrieve the user's choices in your application is to use the Registry class that's part of the FoxPro Foundation classes (Registry in Registry.VCX).

Sound information for applications is stored at HKEY_CURRENT_USER\AppEvents\Schemes\Apps. There are two levels of keys below that. The next is the application itself and the bottom level is the sound name (as it appears in the Sounds applet, unless a name is defined in the EventLabels section described above). Both the application and the individual sounds each need one unnamed value. For the application, it's the name to appear in the Sounds applet. For the individual sounds, it's the WAV file to play.

The SetSounds function shown here (and included in the session materials) adds a list of sounds to the registry, so that the user can choose sound files through the Sounds applet.

```
* Create registry entries for an application and its sounds
LPARAMETERS cInternalName, cAppName, aSoundList
   * cInternalName = the internal name of the application
   *                 - used as the registry key
   * cAppName = the name of the application to appear
   *            in the Sounds applet
   * aSoundList = two-column array - each row contains name
   *              of a sound and the WAV file to play for it.

#DEFINE HKEY_CURRENT_USER -2147483647

* Check parameters
ASSERT VARTYPE(cInternalName) = "C" and NOT EMPTY(cInternalName) ;
   MESSAGE "AddSounds: Must pass cInternalName"
IF VARTYPE(cInternalName) <> "C" OR EMPTY(cInternalName)
   ERROR 11
   RETURN 0
ENDIF

ASSERT VARTYPE(cAppName) = "C" and NOT EMPTY(cAppName) ;
   MESSAGE "AddSounds: Must pass cAppName"
IF VARTYPE(cAppName) <> "C" OR EMPTY(cAppName)
   ERROR 11
   RETURN 0
ENDIF

ASSERT TYPE("aSoundList[1]")="C" ;
```

```
   MESSAGE "AddSounds: Must pass array of sounds"

IF TYPE("aSoundList[1]")<> "C"
   ERROR 11
   RETURN 0
ENDIF

ASSERT ALEN(aSoundList,2) = 2 ;
   MESSAGE "AddSounds: Array must have two columns"

IF ALEN(aSoundList,2) <> 2
   ERROR 230
   RETURN 0
ENDIF

* If we get this far, we have parameters. Still should
* check array contents as we go.

LOCAL oRegisty, cStartKey, cNullVal, nSoundCount, nSound
LOCAL nNewSoundCount

oRegistry = NEWOBJECT("Registry",HOME()+"FFC\Registry")

WITH oRegistry
   cStartKey = "AppEvents\Schemes\Apps"
   * Create a null string
   cNullVal = ""
   cNullVal = .null.

   nNewSoundCount = 0

   IF .IsKey(cStartKey, HKEY_CURRENT_USER)
      * The key we need exists. Go for it.
      * Start by creating the key for the application
      IF .SetRegKey(cNullVal,cAppName,cStartKey + "\" + cInternalName,;
                    HKEY_CURRENT_USER, .t.) = 0

         * Now add the sounds, one by one
         nSoundCount = ALEN(aSoundList, 1)
         FOR nSound = 1 TO nSoundCount
            * Check that both items are provided and that
            * the file exists
            IF TYPE("aSoundList[ nSound, 1]") = "C" ;
               and TYPE("aSoundList[ nSound, 2]") = "C" ;
               and FILE(aSoundList[ nSound, 2])

               * This one looks good, so store the information
               IF .SetRegKey(cNullVal, aSoundList[ nSound, 2], ;
                          cStartKey + "\" + cInternalName + "\" + ;
                          aSoundList[ nSound, 1] + "\.Current", ;
                          HKEY_CURRENT_USER, .t.) = 0
                  nNewSoundCount = nNewSoundCount + 1
               ENDIF
            ENDIF
         ENDFOR
      ELSE
         ERROR "Can't add registry key"
      ENDIF
   ELSE
      ERROR "Registry key does not exist"
   ENDIF
```

```
ENDWITH

RETURN nNewSoundCount
```

To use the function, create a two-column array, putting the names for the sounds in the first column and the default WAV file for each in the second. Then, call the function, like this:

```
DIMENSION aSounds[3, 2]
aSounds[1,1] = "Start"
aSounds[1,2] = "AnnoyApp\Sounds\Startup.WAV"
aSounds[2,1] = "Error"
aSounds[2,2] = "AnnoyApp\Sounds\Buzzer.WAV"
aSounds[3,1] = "End"
aSounds[3,2] = "AnnoyApp\Sounds\Byebye.WAV"
nSoundsAdded = SetSounds( "Annoying","My Annoying Application", @aSounds)
```

The function returns the number of sound items actually stored in the registry, so you can check whether a problem occurred. The session materials include AccDemoSounds.PRG, which registers a few sounds for the demo application.

Once you store the sound information, you can use the sounds in your application by retrieving it from the Registry. Again, the Registry class makes this pretty easy. For example, to extract the sound we just stored for an Error in our application, use code like this:

```
#DEFINE HKEY_CURRENT_USER -2147483647
oRegistry = NEWOBJECT("Registry",HOME()+"FFC\Registry")
cValue = ""
IF .GetRegKey( "", @cValue, ;
               "AppEvents\Schemes\Apps\Annoying\Error\.Current", ;
               HKEY_CURRENT_USER ) = 0
   * Got the sound. Now we can play it.
   DECLARE INTEGER PlaySound IN winmm.dll ;
      STRING cName, INTEGER hModule, INTEGER nFlags
   PlaySound( cValue, 0, 0 )
ENDIF
```

The program PlayAppSound.PRG, included in the session materials, accepts a sound name as parameter, and finds and plays that sound. The materials also include ClearSounds.PRG, which removes the registry entries for all sounds for a specified application, and AccDemoClearSounds.PRG, which uses ClearSounds to remove the entries for the demo application.

Managing fonts

The first font-related thing your application can do for accessibility is make sure that it works properly if the user chooses Large Fonts (or a custom font size) in the Display Properties applet. A surprising number of applications haven't been tested with large fonts and, as a result, have text cut off in some labels and controls when that setting is chosen. (You'll find some examples among the VFP Solutions examples in VFP 6 and earlier versions. Many of them, including the main form of the Solutions application, have display issues with large fonts. These problems have been fixed in VFP 7.)

Dealing with large fonts is pretty simple. Use only scalable fonts. In particular, stay away from MS Sans Serif, a commonly-used non-TrueType font. You can recognize scalable fonts by the TrueType or OpenType logo in the Fonts dialog.

The font users see when entering data in an application is usually controlled within the application. Most applications provide a Font dialog to let the user set the font. In fact, in many applications, the user can set different fonts for different data. For example, in Word, every character can use different settings.

Entering data in a database application is somewhat different than typing a document or entering spreadsheet data, so you probably won't want to allow control at the level that Word does. Nonetheless, user with visual impairments (including older users) will appreciate a way to "bump the font" of your application. In principle, the GetFont() function and the SetAll method make it easy to give the user this control. Here's some code that lets the user choose a font and applies it to a form:

```
* Assume this is in a form-level method
WITH This
   cName = .FontName
   nSize = .FontSize
   lBold = .FontBold
   lItalic = .FontItalic
ENDWITH
IF This.ChangeFont( @cName, @nSize, @lBold, @lItalic)
   THIS.Setall("FontName", cName)
   This.Setall("FontSize", nSize)
   This.Setall("FontBold, lBold)
   This.Setall("FontItalic, lItalic)
ENDIF
RETURN
```

Here's the code for the ChangeFont method:

```
* Let the user choose a font, starting from a specified font
LPARAMETERS tcFontName, tnFontSize, tlFontBold, tlFontItalic

LOCAL cFontName, nFontSize, cStyle

IF PCOUNT() < 4
   RETURN .F.
ELSE
   IF VarType( tcFontName ) = "C"
      cFontName = tcFontName
   ELSE
      cFontName = "Arial"
   ENDIF

   IF VarType( tnFontSize ) <> "N"
      nFontSize = tnFontSize
   ELSE
      nFontSize = 10
   ENDIF

   cStyle = ""
   IF VarType( tlFontBold ) = "L" AND tlFontBold
      cStyle = cStyle + "B"
   ENDIF
   IF VarType( tlFontItalic ) = "L" AND tlFontItalic
```

```
      cStyle = cStyle + "I"
   ENDIF

   * Ask the user for a font
   cFontString = GetFont(cFontName, nFontSize, cStyle)
ENDIF

IF EMPTY(cFontString)
   * User cancelled
   RETURN .F.
ELSE
   * Parse the chosen into its components
   cFontString = CHRTRAN(cFontString, ",", CHR(13))
   ALINES(aFontInfo,cFontString)

   tcFontName = aFontInfo[1]
   tnFontSize = VAL(aFontInfo[2])
   IF "B"$aFontInfo[3]
      tlFontBold = .T.
   ELSE
      tlFontBold = .F.
   ENDIF
   IF "I"$aFontInfo[3]
      tlFontItalic = .T.
   ELSE
      tlFontItalic = .F.
   ENDIF
ENDIF
RETURN .T.
```

However, this approach has several problems. First, it assumes that every text item on the form should use the same font in the same size with the same characteristics. That's unlikely to be the case. There's a second problem as well – enlarging fonts may cause controls on the form to overlap. Finally, once a user sets a font, he's going to expect the application to remember it.

There are solutions to all of these problems. The cusFontHandler class in the session materials allows the user to choose a new font and size and then applies it proportionally to the contained objects. It uses the cusResizer class to enlarge controls as needed and reposition them, again proportionally. (cusResizer is based on the cusResizer class in *1001 Things You Wanted to Know About Visual FoxPro,* by Akins, Kramek and Schummer.)

The session materials also include a set of classes for storing values so they can be restored. The class cusPersistFonts shows how to store the necessary font, size and position information so that the form looks the same when the user opens it again. For demonstration purposes, this class stores the data in a table; you might prefer to store it in the Registry or use another storage mechanism. (cusPersistFonts is a descendent of Doug Hennig's sfPersistent class, also included in the session materials.)

Cooperating with accessibility tools

VFP 7 is far more able to work with accessibility tools than earlier versions of VFP because it supports the IAccessible interface. This means that tools like Magnifier and Narrator can see each individual control on the form. With older versions of VFP, items below the form level weren't visible to such tools. Best of all, you don't have to do anything in your applications to provide this ability.

There are some things you can do, though, that will make it easier for users who work with Accessibility tools.

To aid Narrator and other screen readers, make sure that controls have both informative names and associated labels, where appropriate. In general, screen readers use the Caption property, if it has one, to identify a control.

Their behavior for other controls varies. Narrator uses the control's Name. Some screen readers associate a label with the control and use that to identify it. The label may be picked by its proximity to the control (above it or to the left) or by the tab order. Make sure that you set an appropriate tab order and that your labels for textboxes and editboxes are positioned so that a screen reader will be able to associate them with the right control – in general, that means that the label should be either above or to the left of the control it labels.

Make sure that controls that can have a graphic, like checkboxes and buttons, also include a text Caption. A screen reader can't read a graphics file.

Making forms navigable

It's always a good idea to set the tab order in forms so that it roughly corresponds to the visual order of the form. That's even more important for accessible applications. Focus that jumps all over the form is difficult for users to follow. For someone using an accessibility aid, such jumps can be very confusing.

While users can tab through the controls on a form to navigate, providing hot keys for the controls makes navigation easier, especially for those with motor disabilities. Specify a hot key for a control by putting "\<" before the chosen letter. While not every control has a caption that can take a hot key, VFP provides a trick for those that don't. When a control that can't get focus has a hot key, pressing it sets focus to the next control in the tab order that accepts focus. That is, to give a textbox or editbox a hot key, put a label right before it in the tab order and give the label a hot key.

Navigability isn't just for keyboard users. When laying out a form, keep in mind that some people can use a mouse, but may not have fine control over it. Make controls large enough and space them far enough apart that users with motor difficulties can land on the control they want. (The Windows interface guidelines address the issue of the size and spacing of controls.)

Along the same lines, don't put dangerous commands adjacent to commonly used commands, whether it's in a menu or a toolbar. It's too easy to hit the wrong button or choose the wrong menu item, especially if your motor skills are impaired.

A number of applications, including Visual FoxPro, offer large toolbar buttons as an option. When chosen, the toolbar size increases, as does the size of individual buttons. It's pretty easy to offer this option in your applications. The tbrResize toolbar class, along with the cmdTbrResize button and chkTbrResize checkbox classes, all in Accessibility.VCX in the session materials, provide this functionality. (If you want to use other types of objects in your toolbar, you need to create resizable subclasses for them.) The tbrSample class in Samples.VCX shows an example and the form Options.SCX shows one way you might implement changing the toolbar size. It turns out that the hardest part (at least, most time-consuming) of offering toolbars with both large and small buttons is having the necessary bitmaps (and, where appropriate, bitmap masks) in both sizes. The cusPersistentOptions class in Persistent.VCX remembers the large toolbar setting, so you can restore it when the user returns to the application.

Avoiding timing issues

Some users need more time to read messages or to type input. Don't use messages that disappear after a fixed period of time and don't put a limit on the time a user has to enter data. If an application needs such time limits, make the time period configurable within the application. That is, don't use code like:

```
WAIT WINDOW "Urgent message" TIMEOUT 2
```

Instead, store the timeout period as, say, an application property, providing a way for the user to change it. Then, change the code to:

```
WAIT WINDOW "Urgent message" TIMEOUT oApp.nTimeOut
```

A less crucial timing item, but one that affects the user's perception of your application is the incremental search timing in lists and combos. In VFP 6 and earlier versions, this is tied to the _DBLCLICK system variable. In VFP 7, a new variable, _INCSEEK, controls this setting.

In either case, allowing the user to set the incremental search timeout makes your application easier to use (and not just for users with disabilities). In VFP 7, you can simply assign the user's chosen value to _INCSEEK. In older versions, you need to honor the system double-click setting that's stored in _DBLCLICK, while customizing incremental search. The way to do that is to change _DBLCLICK in the GotFocus of lists and combos and change it back to its normal setting in LostFocus. Since this is needed application-wide, a simple approach is to store the initial _DBLCLICK setting in the application object, along with the user's chosen incremental search setting. Then, changing it is simple:

```
* In a combo or list GotFocus
IF VARTYPE("oApp") <> "U" AND PEMSTATUS(oApp,"nIncSeek",5)
   _DBLCLICK = oApp.nIncSeek
ENDIF

* In a combo or list LostFocus
IF VARTYPE("oApp") <> "U" AND PEMSTATUS(oApp,"nDblClick",5)
   _DBLCLICK = oApp.nDblClick
ENDIF
```

Of course, the best idea is to add this code to your combo and list subclasses.

The incremental search speed stored in _INCSEEK is not a system setting. Allow your users to control this value and remember their settings. The demo application in the system materials shows one way to do this.

Word prediction

For users who find typing difficult, having an application anticipate what they're typing can make life easier. Word prediction works much like Intuit's QuickFill® feature – each time the user types a letter, the application offers the first (or most common) way to complete the string entered so far.

In a VFP application, word prediction makes the most sense when there is a potential list of entries, as with a combobox. The session materials include a set of combo subclasses that provide a QuickFill capability.

There are situations where you may have a list of potential entries for a textbox and a similar approach would be helpful. Consider, for example, a textbox for first names. While the set of possible first names is essentially unlimited, providing word prediction using a list of the 200 or 500 most common first names could cut down the user's typing considerably.

Conclusion

Nearly one in five Americans has at least one disability. Designing software that can be used by the people in that group is wise economically, and in some settings, may be required legally. Creating accessible software with Visual FoxPro is not only possible, but easy, as long as that goal is included from the beginning of the design process.

Acknowledgements

A number of people contributed to this session. Thanks to Marcia Akins and Doug Hennig for providing code, and to Christof Lange, Helmut Lange, Karl Petersen, Scott Finegan and Patrick van Hoorn Alkema for answering questions. Special thanks to my favorite physical therapist, Holly Lankin, for reviewing (and improving) these notes.

Resources

Legal issues

Americans with Disabilities Act home page - http://www.usdoj.gov/crt/ada/adahom1.htm

Q&A on the ADA - http://consumerlawpage.com/brochure/disab.shtml

UK's Disability Discrimination Act - http://www.compactlaw.co.uk/dda95.html

Interface Guidelines

Effective Color Contrast - http://www.lighthouse.org/color_contrast.htm

Guideslines from the Trace Center - http://www.trace.wisc.edu/docs/software_guidelines/software.htm

The Interface Hall of Shame - http://www.iarchitect.com/mshame.htm

Making Text Legible - http://www.lighthouse.org/print_leg.htm

National Center for Accessible Media - http://main.wgbh.org/wgbh/pages/ncam/

Windows Accessibility Features - http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?contentid=28000544

The Windows User Experience - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwue/html/welcome.asp?frame=true, especially Chapter 15

Accessibility sites

Cornucopia of Disability Information - http://codi.buffalo.edu/

Microsoft Accessibility Home - http://www.microsoft.com/enable/

National Information Center for Children and Youth with Disabilities - http://www.nichcy.org/

The Trace Center - http://www.trace.wisc.edu/

Disability statistics

Census Bureau statistics (1991-92) - http://codi.buffalo.edu/graph_based/.demographics/.awd/AWD/AWD.html

Visual impairments - http://www.allaboutvision.com/

Hearing impairments - http://www.zak.co.il/deaf-info/old/home.html, http://deafness.about.com/health/deafness/

Accessibility and the Web

Evaluating web site accessibility - http://www.cast.org/bobby/

Making web pages more accessible - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnacc/html/accssblwebpgs.asp

Overview of Web issues - http://www.useit.com/alertbox/990613.html, http://www.useit.com/alertbox/9610.html