Providing Access For Disabled Users On Open Source Desktops

Malcolm Tredinnick, CommSecure Ltd
<malcolm@commsecure.com.au>

Table of Contents

Abstract

Designing computer software that is usable by everybody, regardless of any physical disabilties, is not a trivial task. However, it can be made much easier with support from standard system and desktop libraries. This paper discusses the design of such a solution in the GNOME desktop and looks at efforts taking place between and within a number of Free and Open Source software projects to provide interoperable solutions for all applications.

1. Introduction

Traditional computer usage scenarios assume that the user will be able to read output on a screen device, enter data via the keyboard, move a pointer to any position on the screen using a mouse or trackpad device and a single, double, or even triple click using the mouse to perform some action in a particular region of the screen.

Sadly, traditional computer usage scenarios are not particularly accomodating of users who cannot do one or more of these things.

In modern society, providing mechanisms to help disabled people perform their standard daily activities is completely normal. Buildings will have entrances suitably equipped for people who cannot climb stairs easily. Pedestrian crossing lights at traffic intersections make a clear noise when it is safe to cross to assist people with visual impairments. Concert halls offer radio transmission devices for people with hearing difficulties. Bathrooms with wheelchair access are available in almost all public facilities. Nobody is surprised by these features

and we encounter them constantly.

Designing and implementing computer software that is usable by everybody requires similar levels of planning to that which has gone into designing the types of physical assistants mentioned in the previous paragraph. At present, the problems involved are fairly well understood and a number of solutions have been implemented. Efforts are underway to ensure that the various solutions are interoperable so that users are not required to make exclusionary choices between applications and developers are not tied to a particular toolkit or implementation.

It appears difficult to give an accurate number of people who require accessibility features to use a computer (at least based on the publically available numbers that the author could locate [1]). However, computer penetration into our daily lives is growing. As time goes on, a larger proportion of disabled people who might otherwise have avoided computer usage will be placed at great inconvencience if they cannot easily use a computer. Further, awareness of this audience is growing. National governments around the globe, including Australia, are writing laws which extend existing anti-discrimination and equal-access laws to clearly include computer-based technology. Software projects which are not considering disabled users today run the risk of being either naturally obsoleted or, more directly, targeted with lawsuits and public investigations in the near future.

In this paper the aim is to break down the problem of creating accessible computer software into a number of separate points that can be addressed independently of each other. This description should make it easier to see how the subsequently discussed implementations of accessible software are indeed attempting to solve the necessary problems. Before proceeding down this path, however, a small note of restraint may be in order: It should be borne in mind as you read through this paper that we are not advocating a degradation of the non-disabled user's experience when using software applications. Modern software is the result of enormous amounts of research into usability problems for everyday, able-bodied users. So although little emphasis is given to these features in the sections that follow, be aware that we are describing and supporting solutions which work *in conjunction* with the normal method of software operation, not as a replacement. In other words, the design of accessible software and computing environments must not be done at the cost of discriminating against the majority of users who are fortunate enough not to be hindered by a disability.

2. Defining The Problem

In order to put the work we are discussing in this paper into the correct perspective, it will be worthwhile to accurately describe the problems we are trying to solve. As described previously, software development in an environment which is not aware of accessibility requirements will make certain assumptions about available input and output methods. A large part of creating accessible applications involves removing the need for these assumptions.

There are accessibility problems when working with computers which are not covered in this section or what follows. These include things such as the design of the various alternative input and output devices we mention. Instead, we are considering solutions that aim to work well with a variety of input and output devices.

For another description of the problem of designing for accessibility, readers may wish to consult [DFD].

2.1. Input and Output

Traditional [2] software requires input via a keyboard and mouse (or trackpad or some other kind of pointing device). Since many users are more comfortable with one or other of these devices and since constantly switching between the two can be inconvenient, many developer guides recommend providing keyboard shortcuts for menu items and buttons and for providing access to all functionality via menus and buttons which can be selected with the pointing device.

We can identify a number of implicit assumptions in the above description. Firstly, from an accessibility viewpoint, we cannot assume too much about the kind of input device that is being used. Not all users will be able to

[1] Readily available Australian statistics point to approximately 19 percent of the population having a disability of some kind. However, in these statistics a *disability* is defined to be an injury or impairment that hinders normal daily operations for a period of six months or longer. Under this definition, a person recovering from a knee reconstruction has a disability. However, they do not necessarily need to utilise any special software when using their computer.

[2] For simplicity, this paper will use the slightly inaccurate term *traditional* to mean software or applications which are usable by able-bodied users and which do not have any particular facilities for disabled users.

use a keyboard, even for only a few tasks. A keyboard may simply not be available. Consider, for example, a partially paralysed person who can manipulate a joystick to control an onscreen pointer, but cannot accurately reach or press keys on a keyboard. Later in this paper (see Dasher in Section 6, “Other Work”), we will see that even the possibility of a two-dimensional control such as a joystick is not necessary and not even a possible assumption in some cases.

For users which can operate a mouse-like device, it may not be possible for them to make fast, accurate multiple button presses, so this needs to be taken into consideration.

It cannot be assumed that mouse input is available. Users with muscular control problems may find it easier to use a keyboard with large keys which provide a larger target area for pressing. We cannot assume that a user will be able to simultaneously hold down more than one key. Alternatively, it is possible that a user may accidently press a key more than once whilst trying to select it — they may not be able to control this action, so it would be nice if the software could make allowances.

As far as output is concerned, most of the problems are related to visual disabilities of various types. Blind users cannot see a screen at all. Their choices for processing output are generally between a braille device (which converts areas of the screen, one line at a time, to braille descriptions) and audio screen readers (which convert areas of the screen to audio descriptions using a speech synthesiser).

Some people are not totally blind, but they do have problems distinguishing objects based on their contrast. Both low-contrast and high-contrast environments can provide problems for different people. Similarly, colour-blindness will prevent some people from being able to distinguish output that a program might provide if it is only presented as the change in colour of some part of the application (for example, indicating an error status by changing an icon from green to red will be impossible for some people to identify). Naturally, small fonts and icons will present a problem to anybody with partial blindness, so that must be accounted for.

Although *most* problems with output are in the visual realm, audio output must be considered as well. Anybody with a hearing difficulty is not going to be be able to respond to prompts or alerts that are only given audibly. I mention this point here for completeness, although it is not usually a problem area in traditional software, as developers have learnt not to assume that everybody has speakers attached to their computer, or that they are working in a quiet environment and not using their speakers to listen to music as they work.

2.2. Information Flow

The “information flow” problem is a result of removing assumptions about the input and output subsystems available to the user. Again, we can start from the design of a traditional application and identify the restrictions imposed by such as setup.

Common applications will present their information in a series of two-dimensional windows. The user will navigate amongst the various buttons, input boxes, labels and text sections in each window, reading the text as they go and entering any information required. The correct information to be entered in each input location is usually indicated by a label in the window or some kind of popup tooltip when the mouse moves over an editable location.

An application may add an implicit third dimension to their information presentation by permitting movement amongst a number of windows or displays. For example, a series of dialog boxes which assist the user in configuring a piece of hardware provides this facility. As does a web page with links off to other web pages. As can be seen from these two examples, the user can be guided through the extra information in either a linear fashion (as in the configuration helper example), or via a more random-access paradigm (the hypertext model). However, in all cases (assuming we are talking about a well-designed application), fragments of information that are related to each other will appear near each other as the user perceives them. This avoids forcing the user to skip back and forth between areas of the application to utilise related features and is a standard part of most human interface guidelines.

All of this reasoning remains valid when thinking about accessible software. Only now we must remember that items which appear local to each other in a two-dimensional visual layout may lose that association when viewed in the more linear fashion that a speech synthesiser or line-based braille device presents them. Similarly, somebody who needs to use a much larger size for all of the onscreen elements (due to a partial visual impairment) may not be able to see all of the contents of a large dialog box at once. So a full-screen window for an able-bodied person using a 1024x768 sized screen is going to be be spread over many screens when viewed through an output device that magnifies the size of each normal pixel. Suddenly a label in the top-left corner re-

ferring to something in the bottom-right is no longer a local reference.

Possibly less obvious is the problem of how do we know which text label refers to a given entry box. Spatially, the local arrangement is obvious — this may be lost when viewed through other mechanisms. And tooltips which only appear when the mouse moves over a widget need to be addressed. It has already been pointed out that not everybody uses a mouse and not everybody who does use a mouse can necessarily position it accurately and steadily enough to read the tooltip.

2.3. Starting The User's Session

Consider a scenario involving a public-access terminal in a library or Internet kiosk. Typically, the user only needs to log into the terminal or move the mouse or press a key to deactivate a screen saver and they can being to use the machine.

A similar situation, even for a machine that is running a full complement of accessible software, is more difficult. The generic problem is how to tell which input method a user is going to be most comfortable with. This needs to be done in a way that is achievable by all users; it is no good requesting input via the keyboard in answer to a question such as “Are you able to use the keyboard? (Yes or No)”

A related difficulty is installing a machine or using it for the first time. Inserting a CD containing the latest release of one's favourite Linux distribution and moving through the installation process is a simple task for able-bodied users. Ideally, convincing the installation process that you cannot see the screen and need to use an external speech synthesiser to do the installation should also be possible.

In general, the bootstrapping problem is difficult to solve and practical, complete solutions are not yet available. However, that just makes the issue more pressing; it is an early obstacle encountered by disabled users and one that is usually impossible to avoid.

3. General Solutions

Before proceeding to illustrate an implementation of an accessibility infrastructure (see Section 4, “The GNOME Implementation”), it will be beneficial to outline what a generic solution to the above problems looks like. The ideas described in this section are essentially an abstracted version of the solutions implemented in a number of existing desktop environments. They may not be the only solutions, but experience has shown that they fulfill requirements nicely and are practical to implement.

3.1. Input

Despite the apparent variability in input possibilities (as seen in the previous section), they can be broken down into two main groups: input devices that can be made to present keyboard-like input to applications and input devices that can be translated to look like mouse input to applications.

Consequently, in order to ease the burden on individual application developers, it pays to separate the input layer(s) from the main applications. Each application must be designed so that it can be operated entirely via a keyboard-like device or entirely via a mouse. A translation layer is responsible for taking input from each connected input device and sending the appropriate keyboard or mouse events to the application currently receiving input. If these events are delivered in a "clever enough" fashion — so that they look exactly like normal X events, for example — applications need never know that they are talking to a special input device.

Creating this separation between specific input devices and applications means that different groups of developers can work in the areas of their speciality. Application developers need only follow the rules about requiring a single input device (mouse or keyboard). Input driver writers only need to focus on delivering keystrokes or mouse events in a standardised, well-documented fashion. The two groups can work independently.

There will be some applications that cannot possibly work without the requiring some alphanumeric input from the user. This may appear to reintroduce the assumption that a keyboard is available. However, it really just introduces the requirement that an input assisant of some kind is available which can turn mouse events into keyboard events. Typically, this is some kind of onscreen keyboard application where the user selects the appropriate keys with the mouse pointer (see also the Dasher description in Section 6, “Other Work”).

Note that in practice, input device drivers do not *just* turn input events into events suitable for passing to an ap-

plication. There will often be special control events that must be interpreted as being specific to the type of input device, or that are control events for the current output method (for example, switching the ouput mode from reading the contents of a particular window to reviewing the entire screen contents). However, such “out of band” messages are really an implementation detail (although this does not diminish their importance or difficulty) and should not interfere with the ideal solution model being described here.

3.2. Output

Providing output in an accessibile fashion is a little more difficult than input. This is primarily because there are more types of output events than input events. Output events can involve drawing text to an area of the screen, drawing a graphic, changing the active window, focusing a new button, changing the colour of a region of the screen, and so on. All of these events may be significant to the user's experience and so there needs to be some way of reporting their occurrence via the configured output device.

Negotiating around some of the output assumptions described in Section 2.1, “Input and Output” is relatively straightforward. These are the requirements of users who have partial, if not total, visual acuity (and who may or may not have a hearing impairment). Applications should not rely solely on audio output to convey information. If an application needs to use colours to indicate state, it must be configurable as to the colour palette it uses. There should at least be the possibility of using a high- or low-contrast palette. Ideally, purely colourised prompts should be avoided where possible and distinguishable icons used to represent different states. Whilst on the topic of icons, they should not be too detailed, lest a user who can only perceive their outline be confused as to the purpose of two similarly shaped icons. Font sizes and families should be configurable to accomodate users requiring large print versions of an application.

For users requiring alternative output devices (some kind of screen reader), extra support is required from the desktop. Each of the output events described above must be recognisable and converted into an appropriate message for the screen reader. This is solved by having each *widget* (button, label, text area, drawing area, icon) in the application being able to send an appropriate message when it changes state in some way — either because its content has changed, or it has received or lost the input focus, or for any other reason. An external application will be listening for all such events from every application that is accessible and then sending them to the current output device.

In this way, a collection of more or less random on-screen events are converted into a set of standard events by the listener application and then it is up to each output device driver to convert these more-or-less standard events into an appropriate form for the output device. Example of an *appropriate form* here might be a braille representation of the text version of each message for a braille-based screen reader, or a vocalisation of each message — possibly in a different voice depending upon the type of message — for a speech synthesiser device.

This method of presenting output in a device-independent fashion does place some responsibility on the shoulders of the application developer. In the input case, application authors did not really need to be aware that their input may be coming from a special device. For output, however, if they have created a non-standard widget of some kind, the author needs to also implement the appropriate accessibility messages for that widget. Otherwise, any action in the widget will not generate messages that can be passed to specialised output devices. By and large, this extra requirement is not too onerous. Most of the active areas in most applications are made up from standard widgets. Only rarely will an author be creating an entirely new type of widget and in only those cases will he need to implement the accessibility methods (which are usually fairly routine in practice).

3.3. Information Flow

The information flow problem outlined in Section 2.2, “Information Flow” is a problem of *referring to related data*. That is to say, an application needs to provide clues as to which pieces of information belong together. From an accessibility solution point of view, this can be solved by attaching a piece of information to many widgets describing what other widget (if any) they are referring to and where they reside in the hierarchy of widgets. For example, a label widget should include information about which input box or text area it is labelling. Each focusable widget should be explicitly given an index to specify the order in which the widgets are highlighted when the “move to next widget” key (often the **Tab** key) is pressed.

Even for non-disabled users, application designers are encouraged to order the focus order of widgets in a logical fashion. It assists nobody if the focus jumps around a window of widgets with no apparent order. Precisely the same guidelines apply when designing with accessibility requirements in mind. If the user needs to enter an address, the input boxes should logically ask for the street name, suburb, postcode, country and so on in a sensible order. Filling in an address by completing the street, then the postcode, then the country, then the town and

finally the addressee's name is tortuous and error-prone.

4. The GNOME Implementation

We will now consider a practical application of the information from the previous two sections: the implementation of the accessibility framework in the GNOME desktop [http://www.gnome.org/]. This particular choice of example is chosen because of the present author's familiarity with the project, rather than because it is necessarily the best implementation in any sense [3]. This section can only be an overview of what exists today in GNOME. Readers interested in the details (particularly of a technical nature) should refer to [GAP] for more content.

The GNOME accessibililty implementation is designed along the lines outlined in Section 3, “General Solutions”. Acknowledging that one configuration cannot possibly suit all disabled and non-disabled users, the accessibility tools are highly configurable and, when not required, they consume no resources at all.

A standard GNOME installation comes with a number of accessibility input and output devices. For input, an onscreen keyboard is supplied (permitting users who can only use a mouse to enter alphanumeric data). Also, configuration options exist for sticky keys (enabling a user to enter a key combination such as **Shift**-**A** by first pressing and releasing **Shift** and then pressing and releasing **A**), slow keys (only accept keys that are held down for a certain amount of time), debounced keys (ignore duplicate keys that occur within a certain amount of time of the previous occurrence), and obviously the mouse can be configured to control how fast the pointer moves and whether the mouse pointer should be highlighted specially when a key is pressed (to assist users in locating the pointer).

Commonly available output options for accessibility purposes in GNOME include interfaces to a number of common speech synthesisers, including the freely available Festival and FreeTTS packages. An onscreen magnifier if also available, which turns half of the user's screen into a highly magnified image of the area around the mouse pointer in the other half of the screen. An onscreen braille simulator is also available, although clearly this is of no use to people requiring braille output. It is, however, of some use to developers (at least, for those developers who can read braille).

As proposed in the general solution, the input and output device layers are separated from each individual application. Interaction between the input/output layers and applications is done via the *Accessibility Toolkit Service Provider Interface* (AT-SPI). When the desktop is running with the accessibility features enabled, a registry daemon runs with which both applications and I/O devices register their availability. Events are then passed back and forth between the devices and the appropriate applications via the AT-SPI. At the lowest level, the AT-SPI implementation uses CORBA to provide a standardised communications protocol between disparate applications. Because of this design, bridges to other applications can also be built that interoperate nicely with the GNOME infrastructure. As proof of this, a bridge to permit Java applications to automatically be accessible via GNOME-based input and output devices has been written and is in wide use.

The previous section explained how an accessible infrastructure would require cooperation from the widgets making up an application's interface in order to provide appropriate output. In GNOME, the basic widget set is provided by GTK+ (the GIMP Toolkit — see [GTK]). Supporting the GTK+ library is the ATK library (Accessibility Toolkit), which provides the interfaces used by the accessibility layer to communicate between applications and input/output devices.

In this design, every widget that is available in the GTK+ library is automatically accessible. The application developer does need to do a little bit of work in order to set the appropriate descriptions for each widget, but the bulk of the work is already done in the low-level platform libraries. To see that setting descriptions is necessary (and cannot possibly be automated), imagine an address entry widget. This will typically contain a number of more or less identical looking input boxes. However, one box will be for the name of the addressee, one for the street address, one for the suburb and so on. Attaching appropriate descriptions to each widget will enable a screen reader to provide relevant positional information to the user.

For developers who create detailed custom widgets, extensive documentation on both the design and API levels are available ([GAD], [ATK]).

To assist with the problems introduced with contrast resolution, colour blindness, font sizes and so forth, GNOME has a very comprehensive theme architecture in place. All aspects of the desktop, from the GTK+ wid-

[3] Although in terms of completeness and functionality, my opinion is that the GNOME implementation is a leading example of what can be accomplished.

gets, to the window manager, to standard icons used by many applications are themable. The GNOME accessibility documentation and the human interface guidelines ([HIG]) strongly suggest that applications cooperate with the theme supplied by the platform libraries and do not try to specifically override features to suit the designer's whims. A standard GNOME installation comes with a number of themes, including low- and high-contrast versions and a large print version, which comes in normal, low- and high-contrast variants. The theme is set in a single place (via the desktop preferences) and thus uniformly configures the entire user experience.

The bootstrapping problem mentioned earlier (see Section 2.3, “Starting The User's Session”) is an ongoing project in GNOME. Some support is available, but it is not nearly complete. Presently, a user who requires assistive technology may require some help from another person in order to initially configure their environment to a usable state. This configuration is preserved across sessions for the same user, but this obviously does not solve the problem of having an accessible terminal in a university laboratory, an Internet kiosk or a public library.

4.1. Assessing Accessible Applications

Providing a supporting infrastructure for developers does not automatically create accessibile applications. Although most of the required steps are fairly automatic, it is not difficult to omit to create a description on some widgets, or to not link a label widget to its associated input box, or to forget to add a keyboard accelerator to a menu item, or to create a widget that does not completely obey the desktop theme.

Ensuring that applications meet the GNOME accessibility guidelines is, at present, a mostly manual task. Wipro, an Indian company working with Sun Microsystems, have written a test plan that quality assurance testers can follow to ascertain an application's compliance ([Tests]). Work has also begun on creating a tool which would automate some of these tests and highlight areas of potential trouble. Similar tools already exist for testing web page accessibility and for testing the accessibility features in Java applications. Such a tool cannot completely replace a manual QA process, but it can help speed up the process and provide a first-pass for a developer before they pass on any questions or problems to a more experienced accessibility developer.

5. The Free Standards Group Accessibility Workgroup

In an effort to coordinate the development and popularisation of accessibility solutions that work well together, an Accessibility Workgroup ([FSG]) was formed in February 2003. This workgroup is a member of the Free Standards Group [http://ww.freestandards.org/], which amongst other things oversees the Linux Standards Base efforts and some cross-platform internationalisation efforts. The Free Standards Group is maybe not as widely known as they should be, but their various workgroups (including the accessibility group) are tackling some large, difficult standardisation efforts.

The accessibility workgroup is a group of 18 core members, plus a couple of advisors who create subcommittees to concentrate on various areas of accessibility interoperability. Currently, there are committees for keyboard accessibility, AT-SPI, and AT shared-device IO. This last committee is focusing on problems such as defining interfaces so that speech synthesisers can work nicely with other audio device users (for example, a vocal description of a window should not have to wait until the current song has finished playing).

The workgroup charter ([FSG-Charter]) is worth reading, as it outlines the workgroup's plans for the next couple of years (with slightly more vague goals for the years after that). Developers implementing accessibility infrastructure and administrators and procurement staff responsible for rolling out accessibile systems will benefit from following the efforts of this workgroup. The minutes of the weekly telephone conferences are promptly made available and provide a good overview of what is happening in this rapidly developing area.

6. Other Work

Not surprisingly, GNOME is not the only desktop project providing accessibility support. The KDE project [http://www.kde.org/] has been working on a similar infrastructure for their applications. Their progress can be seen at [KDE]. At present, KDE is a little behind GNOME on this front, since the accessibility support has only recently appeared inside their toolkit. Interoperability between KDE and GNOME, two of the major open source desktops, is a slight problem at the moment. It is theoretically possible for KDE applications to interact with screen readers and other assistive applications written for the GNOME AT-SPI implementation. However, since

KDE does not use CORBA for interprocess communications, this particular implementation appears to add an extra requirement for any applications that wish to do this.

On a wider front, [Linux] tries to cover a broad spread of accessibility projects that concern Linux and its distributions. This is a good place to start to get an overview of general accessibility support on Linux.

Dasher (http://www.inference.phy.cam.ac.uk/dasher/) is an innovative project to provide a new input method. The Dasher project's website contains an animated image which best describes how the tool works, but it essentially involves the user *steering* the pointer towards the next character in the input they are creating as the new characters stream in from the right hand side of the input box. For people who can only manipulate a mouse, possibly even controlled by their foot, this input method can be surprisingly quick and accurate. The testimonials on the website show that Dasher has fulfilled a need and deserves the attention it is getting.

Resources

[ATK] *Accessibility Toolkit API documentation [http://developer.gnome.org/doc/API/2.0/atk/atk.html]*.

[DFD] *Designing For Disabilities [http://developer.gnome.org/projects/gap/hi-design.html]*.

[FSG] *Free Standards Group Accessibility Workgroup [http://accessibility.freestandards.org/index.php]*.

[FSG-Charter] *Accessibility Workgroup Charter [http://accessibility.freestandards.org/modules.php?name=Content&pa=showpage&pid=20]*.

[GAD] *GNOME Accessibility For Developers [http://developer.gnome.org/projects/gap/guide/gad/index.html]*.

[GAP] *The GNOME Accessibility Project [http://developer.gnome.org/projects/gap/]*.

[GTK] *GTK+ [http://www.gtk.org/] — The GIMP Toolkit*.

[HIG] *GNOME Human Interface Guidelines [http://developer.gnome.org/projects/gup/hig/]*.

[KDE] *KDE Accessibility Project [http://accessibility.kde.org/]*.

[Linux] *Linux Accessibility Resources [http://trace.wisc.edu/linux/]*.

[Tests] *Testing GNOME Applications For Accessibility [http://developer.gnome.org/projects/gap/testing/index.html]*.