Writing Programs Using

newt

Erik Troan, <ewt@redhat.com>

v0.31, 2003-Jan-06

The newt windowing system is a terminal-based window and widget library designed for writing applications with a simple, but user-friendly, interface. While newt is not intended to provide the rich feature set advanced applications may require, it has proven to be flexible enough for a wide range of applications (most notably, Red Hat's installation process). This tutorial explains the design philosophy behind newt and how to use newt from your programs.

1. Introduction

Newt has a definite design philosophy behind it, and knowing that design makes it significantly easier to craft robust newt applications. This tutorial documents newt 0.30 --- older versions of newt had annoying inconsistencies in it (which writing this tutorial pointed out), which were removed while this tutorial was written. The latest version of newt is always available from Red Hat.

1.1. Background

Newt was originally designed for use in the install code for Red Hat Linux. As this install code runs in an environment with limited resources (most importantly limited filesystem space), newt's size was immediately an issue. To help minimize its size, the following design decisions were made early in its implementation:

- newt does not use an event-driven architecture.
- newt is written in C, not C++. While there has been interest in constructing C++ wrapper classes around the newt API, nothing has yet come of those ideas.
- Windows must be created and destroyed as a stack (in other words, all newt windows behave as modal dialogs). This is probably the greatest functionality

restriction of `newt`.

- The tty keyboard is the only supported input device.
- Many behaviours, such as widget traversal order, are difficult or impossible to change.

While `newt` provides a complete API, it does not handle the low-level screen drawing itself. Instead, `newt` is layered on top of the screen management capabilities of John E. Davis's S-Lang (ftp://space.mit.edu/pub/davis/slang/) library.

1.2. Designing `newt` applications

As `newt` is not event driven and forces modal windows (forcing window order to behave like a stack), newt applications tend to look quite like other text-mode programs. It is quite straightforward to convert a command line program which uses simple user prompts into a `newt` application. Some of the programs run as part of the Red Hat installation process (such as `Xconfigurator` and `mouseconfig`) were originally written as simple terminal mode programs which used line-oriented menus to get input from the user and were later converted into `newt` applications (through a process affectionately known as newtering). Such a conversion does not require changes to the control flow of most applications. Programming `newt` is dramatically different from writing programs for most other windowing systems as `newt`'s API is not event driven. This means that `newt` applications look dramatically different from programs written for event-driven architectures such as Motif, `gtk`, or even Borland's old TurboVision libraries. When you're designing your `newt` program, keep this differentiation in mind. As long as you plan your application to call a function to get input and then continue (rather then having your program called when input is ready), programming with the newt libraries should be simple.

1.3. Components

Displayable items in `newt` are known as *components*, which are analogous to the widgets provided by most Unix widget sets. There are two main types of components in `newt`, forms and everything else. Forms logically group components into functional sets. When an application is ready to get input from a user, it "runs a form", which makes the form active and lets the user enter information into the components the form contains. A form may contain any other component, including other forms. Using subforms in this manner lets the application change the details of how the user tabs between components on the form, scroll regions of the screen, and control background colors for portions of windows. Every component is of type `newtComponent`, which is an opaque type. It's guaranteed to be a pointer though,

which lets applications move it through void pointers if the need arises. Variables of type `newtComponent` should never be directly manipulated -- they should only be passed to `newt` functions. As `newtComponent` variables are pointers, remember that they are always passed by value -- if you pass a `newtComponent` to a function which manipulates it, that component is manipulated everywhere, not just inside of that function (which is nearly always the behaviour you want).

1.4. Conventions

`Newt` uses a number of conventions to make it easier for programmers to use.

- All functions which manipulate data structures take the data structure being modified as their first parameter. For example, all of the functions which manipulate forms expect the `newtComponent` for that form to be the first parameter.
- As `newt` is loosely typed (forcing all of the components into a single variable makes coding easier, but nullifies the value of type checking), `newt` functions include the name of the type they are manipulating. An example of this is `newtFormAddComponent()`, which adds a component to a form. Note that the first parameter to this function is a form, as the name would suggest.
- When screen coordinates are passed into a function, the x location precedes the y location. To help keep this clear, we'll use the words "left" and "top" to describe those indicators (with left corresponding to the x position).
- When box sizes are passed, the horizontal width precedes the vertical width.
- When both a screen location and a box size are being passed, the screen location precedes the box size.
- When any component other then a form is created, the first two parameters are always the (left, right) location.
- Many functions take a set of flags as the final parameter. These flags may be logically ORed together to pass more then one flag at a time.
- `Newt` uses *callback* functions to convey certain events to the application. While callbacks differ slightly in their parameters, most of them allow the application to specify an arbitrary argument to be passed to the callback when the callback is invoked. This argument is always a `void *`, which allows the application great flexibility.

2. Basic `Newt` Functions

While most `newt` functions are concerned with widgets or groups of widgets (called grids and forms), some parts of the `newt` API deal with more global issues, such as initializing `newt` or writing to the root window.

2.1. Starting and Ending `newt` Services

There are three functions which nearly every `newt` application use. The first two are used to initialize the system.

```
int newtInit(void);
void newtCls(void);
```

`newtInit()` should be the first function called by every `newt` program. It initializes internal data structures and places the terminal in raw mode. Most applications invoke `newtCls()` immediately after `newtInit()`, which causes the screen to be cleared. It's not necessary to call `newtCls()` to use any of `newt`'s features, but doing so will normally give a much neater appearance. When a `newt` program is ready to exit, it should call `newtFinished()`.

```
int newtFinished(void);
```

`newtFinished()` restores the terminal to its appearance when `newtInit()` was called (if possible -- on some terminals the cursor will be moved to the bottom, but it won't be possible to remember the original terminal contents) and places the terminal in its original input state. If this function isn't called, the terminal will probably need to be reset with the `reset` command before it can be used easily.

2.2. Handling Keyboard Input

Normally, `newt` programs don't read input directly from the user. Instead, they let `newt` read the input and hand it to the program in a semi-digested form. `Newt` does provide a couple of simple functions which give programs (a bit of) control over the terminal.

```
void newtWaitForKey(void);
void newtClearKeyBuffer(void);
```

The first of these, `newtWaitForKey()`, doesn't return until a key has been pressed. The keystroke is then ignored. If a key is already in the terminal's buffer, `newtWaitForKey()` discards a keystroke and returns immediately.

`newtClearKeyBuffer()` discards the contents of the terminal's input buffer without waiting for additional input.

2.3. Drawing on the Root Window

The background of the terminal's display (the part without any windows covering it) is known as the *root window* (it's the parent of all windows, just like the system's root directory is the parent of all subdirectories). Normally, applications don't use the root window, instead drawing all of their text inside of windows (`newt` doesn't require this though -- widgets may be placed directly on the root window without difficulty). It is often desirable to display some text, such as a program's name or copyright information, on the root window, however. `Newt` provides two ways of displaying text on the root window. These functions may be called at any time. They are the only `newt` functions which are meant to write outside of the current window.

```
void newtDrawRootText(int left, int top, const char * text);
```

This function is straightforward. It displays the string `text` at the position indicated. If either the `left` or `top` is negative, the position is measured from the opposite side of the screen. The final measurement will seem to be off by one though. For example, a `top` of -1 indicates the last line on the screen, and one of -2 is the line above that. As it's common to use the last line on the screen to display help information, `newt` includes special support for doing exactly that. The last line on the display is known as the *help line*, and is treated as a stack. As the value of the help line normally relates to the window currently displayed, using the same structure for window order and the help line is very natural. Two functions are provided to manipulate the help line.

```
void newtPushHelpLine(const char * text);
void newtPopHelpLine(void);
```

The first function, `newtPushHelpLine()`, saves the current help line on a stack (which is independent of the window stack) and displays the new line. If `text` is `NULL`, `newt`'s default help line is displayed (which provides basic instructions on using `newt`). If `text` is a string of length 0, the help line is cleared. For all other values of `text`, the passed string is displayed at the bottom, left-hand corner of the display. The space between the end of the displayed string the the right-hand edge of the terminal is cleared. `newtPopHelpLine()` replaces the current help line with the one it replaced. It's important not to call tt/newtPopHelpLine()/ more then `newtPushHelpLine()`! `Suspending Newt Applications` By default, `newt` programs cannot be suspended by the user (compare this to most Unix programs which can be suspended by pressing the suspend key (normally `^Z`). Instead,

programs can specify a *callback* function which gets invoked when the user presses the suspend key.

```
typedef void (*newtSuspendCallback)(void);

void newtSetSuspendCallback(newtSuspendCallback cb);
```

The suspend function neither expects nor returns any value, and can do whatever it likes to when it is invoked. If no suspend callback is registered, the suspend keystroke is ignored. If the application should suspend and continue like most user applications, the suspend callback needs two other `newt` functions.

```
void newtSuspend(void);
void newtResume(void);
```

`newtSuspend()` tells `newt` to return the terminal to its initial state. Once this is done, the application can suspend itself (by sending itself a `SIGTSTP`, fork a child program, or do whatever else it likes. When it wants to resume using the `newt` interface, it must call `newtResume` before doing so. Note that suspend callbacks are not signal handlers. When `newtInit()` takes over the terminal, it disables the part of the terminal interface which sends the suspend signal. Instead, if `newt` sees the suspend keystroke during normal input processing, it immediately calls the suspend callback if one has been set. This means that suspending newt applications is not asynchronous.

2.4. Refreshing the Screen

To increase performance, S-Lang only updates the display when it needs to, not when the program tells S-Lang to write to the terminal. “When it needs to” is implemented as “right before the we wait for the user to press a key”. While this allows for optimized screen displays most of the time, this optimization makes things difficult for programs which want to display progress messages without forcing the user to input characters. Applications can force S-Lang to immediately update modified portions of the screen by calling `newtRefresh`.

1. The program wants to display a progress message, without forcing for the user to enter any characters.
2. A misfeature of the program causes part of the screen to be corrupted. Ideally, the program would be fixed, but that may not always be practical.

2.5. Other Miscellaneous Functions

As always, some function defy characterization. Two of `newt`'s general function fit this oddball category.

```
void newtBell(void);
void newtGetScreenSize(int * cols, int * rows);
```

The first sends a beep to the terminal. Depending on the terminal's settings, this been may or may not be audible. The second function, `newtGetScreenSize()`, fills in the passed pointers with the current size of the terminal.

2.6. Basic `newt` Example

To help illustrate the functions presented in this section here is a short sample `newt` program which uses many of them. While it doesn't do anything interesting, it does show the basic structure of `newt` programs.

```
#include <newt.h>
#include <stdlib.h>

int main(void) {
    newtInit();
    newtCls();

    newtDrawRootText(0, 0, "Some root text");
    newtDrawRootText(-25, -2, "Root text in the other corner");

    newtPushHelpLine(NULL);
    newtRefresh();
    sleep(1);

    newtPushHelpLine("A help line");
    newtRefresh();
    sleep(1);

    newtPopHelpLine();
    newtRefresh();
    sleep(1);

    newtFinished();
}
```

3. Windows

While most `newt` applications do use windows, `newt`'s window support is actually extremely limited. Windows must be destroyed in the opposite of the order they were created, and only the topmost window may be active. Corollaries to this are:

- The user may not switch between windows.
- Only the top window may be destroyed.

While this is quite a severe limitation, adopting it greatly simplifies both writing `newt` applications and developing `newt` itself, as it separates `newt` from the world of event-driven programming. However, this tradeoff between function and simplicity may make `newt` unsuitable for some tasks.

3.1. Creating Windows

There are two main ways of opening `newt` windows: with or without explicit sizings. When grids (which will be introduced later in this tutorial) are used, a window may be made to just fit the grid. When grids are not used, explicit sizing must be given.

```
int newtCenteredWindow(int width, int height, const char * title);
int newtOpenWindow(int left, int top, int width, int height,
     const char * title);
```

The first of these functions open a centered window of the specified size. The `title` is optional -- if it is `NULL`, then no title is used. `newtOpenWindow*(` is similar, but it requires a specific location for the upper left-hand corner of the window.

3.2. Destroying Windows

All windows are destroyed in the same manner, no matter how the windows were originally created.

```
void newtPopWindow(void);
```

This function removes the top window from the display, and redraws the display areas which the window overwrote.

4. Components

Components are the basic user interface element `newt` provides. A single component may be (for example) a listbox, push button checkbox, a collection of other components. Most components are used to display information in a window, provide a place for the user to enter data, or a combination of these two functions. Forms, however, are a component whose primary purpose is not noticed by the user at all. Forms are collections of components (a form may contain another form) which logically relate the components to one another. Once a form is created and had all of its constituent components added to it, applications normally then run the form. This gives control of the application to the form, which then lets the user enter data onto the form. When the user is done (a number of different events qualify as "done"), the form returns control to the part of the application which invoked it. The application may then read the information the user provided and continue appropriately. All `newt` components are stored in a common data type, a `newtComponent` (some of the particulars of `newtComponent`s have already been mentioned. While this makes it easy for programmers to pass components around, it does force them to make sure they don't pass entry boxes to routines expecting push buttons, as the compiler can't ensure that for them. We start off with a brief introduction to forms. While not terribly complete, this introduction is enough to let us illustrate the rest of the components with some sample code. We'll then discuss the remainder of the components, and end this section with a more exhaustive description of forms.

4.1. Introduction to Forms

As we've mentioned, forms are simply collections of components. As only one form can be active (or running) at a time, every component which the user should be able to access must be on the running form (or on a subform of the running form). A form is itself a component, which means forms are stored in `newtComponent` data structures.

```
newtComponent newtForm(newtComponent vertBar, const char * help, int flags
```

To create a form, call `newtForm()`. The first parameter is a vertical scrollbar which should be associated with the form. For now, that should always be `NULL` (we'll discuss how to create scrolling forms later in this section). The second parameter, `help`, is currently unused and should always be `NULL`. The `flags` is normally 0, and other values it can take will be discussed later. Now that we've waved away the complexity of this function, creating a form boils down to simply:

```
newtComponent myForm;

myForm = newtForm(NULL, NULL, 0);
```

After a form is created, components need to be added to it --- after all, an empty form isn't terribly useful. There are two functions which add components to a form.

```
void newtFormAddComponent(newtComponent form, newtComponent co);
void newtFormAddComponents(newtComponent form, ...);
```

The first function, `newtFormAddComponent()`, adds a single component to the form which is passed as the first parameter. The second function is simply a convenience function. After passing the form to `newtFormAddComponents()`, an arbitrary number of components is then passed, followed by `NULL`. Every component passed is added to the form. Once a form has been created and components have been added to it, it's time to run the form.

```
newtComponent newtRunForm(newtComponent form);
```

This function runs the form passed to it, and returns the component which caused the form to stop running. For now, we'll ignore the return value completely. Notice that this function doesn't fit in with `newt`'s normal naming convention. It is an older interface which will not work for all forms. It was left in `newt` only for legacy applications. It is a simpler interface than the new `newtFormRun()` though, and is still used quite often as a result. When an application is done with a form, it destroys the form and all of the components the form contains.

```
void newtFormDestroy(newtComponent form);
```

This function frees the memory resources used by the form and all of the components which have been added to the form (including those components which are on subforms). Once a form has been destroyed, none of the form's components can be used.

4.2. Components

Non-form components are the most important user-interface component for users. They determine how users interact with `newt` and how information is presented to them.

4.3. General Component Manipulation

There are a couple of functions which work on more then one type of components. The description of each component indicates which (if any) of these functions are valid for that particular component.

```
typedef void (*newtCallback)(newtComponent, void *);
```

```
void newtComponentAddCallback(newtComponent co, newtCallback f, void * dat
void newtComponentTakesFocus(newtComponent co, int val);
```

The first registers a callback function for that component. A callback function is a function the application provides which `newt` calls for a particular component. Exactly when (if ever) the callback is invoked depends on the type of component the callback is attached to, and will be discussed for the components which support callbacks. `newtComponentTakesFocus()` works on all components. It allows the application to change which components the user is allowed to select as the current component, and hence provide input to. Components which do not take focus are skipped over during form traversal, but they are displayed on the terminal. Some components should never be set to take focus, such as those which display static text.

4.4. Buttons

Nearly all forms contain at least one button. `Newt` buttons come in two flavors, full buttons and compact buttons. Full buttons take up quit a bit of screen space, but look much better then the single-row compact buttons. Other then their size, both button styles behave identically. Different functions are used to create the two types of buttons.

```
newtComponent newtButton(int left, int top, const char * text);
newtComponent newtCompactButton(int left, int top, const char * text);
```

Both functions take identical parameters. The first two parameters are the location of the upper left corner of the button, and the final parameter is the text which should be displayed in the button (such as “Ok” or “Cancel”).

4.4.1. Button Example

Here is a simple example of both full and compact buttons. It also illustrates opening and closing windows, as well a simple form.

```
#include <newt.h>
#include <stdlib.h>

void main(void) {
    newtComponent form, b1, b2;
    newtInit();
    newtCls();

    newtOpenWindow(10, 5, 40, 6, "Button Sample");
```

```
    b1 = newtButton(10, 1, "Ok");
    b2 = newtCompactButton(22, 2, "Cancel");
    form = newtForm(NULL, NULL, 0);
    newtFormAddComponents(form, b1, b2, NULL);

    newtRunForm(form);

    newtFormDestroy(form);
    newtFinished();
}
```

4.5. Labels

Labels are `newt`'s simplest component. They display some given text and don't allow any user input.

```
newtComponent newtLabel(int left, int top, const char * text);
void newtLabelSetText(newtComponent co, const char * text);
```

Creating a label is just like creating a button; just pass the location of the label and the text it should display. Unlike buttons, labels do let the application change the text in the label with `newtLabelSetText`. When the label's text is changed, the label automatically redraws itself. It does not clear out any old text which may be leftover from the previous time is was displayed, however, so be sure that the new text is at least as long as the old text.

4.6. Entry Boxes

Entry boxes allow the user to enter a text string into the form which the application can later retrieve.

```
typedef int (*newtEntryFilter)(newtComponent entry, void * data, int ch,
          int cursor);

newtComponent newtEntry(int left, int top, const char * initialValue, int
   char ** resultPtr, int flags);
void newtEntrySet(newtComponent co, const char * value, int cursorAtEnd);
char * newtEntryGetValue(newtComponent co);
void newtEntrySetFilter(newtComponent co, newtEntryFilter filter, void * d
```

`newtEntry()` creates a new entry box. After the location of the entry box, the initial value for the entry box is passed, which may be `NULL` if the box should start off empty. Next, the width of the physical box is given. This width may or may not limit the length of the string the user is allowed to enter; that depends on the `flags`. The `resultPtr` must be the address of a `char *`. Until the entry box is destroyed by `newtFormDestroy()`, that `char *` will point to the current value of the entry box. It's important that applications make a copy of that value before destroying the form if they need to use it later. The `resultPtr` may be `NULL`, in which case the user must use the `newtEntryGetValue()` function to get the value of the entry box. Entry boxes support a number of flags:

NEWT_ENTRY_SCROLL

> If this flag is not specified, the user cannot enter text into the entry box which is wider then the entry box itself. This flag removes this limitation, and lets the user enter data of an arbitrary length.

NEWT_FLAG_HIDDEN

> If this flag is specified, the value of the entry box is not displayed. This is useful when the application needs to read a password, for example.

NEWT_FLAG_RETURNEXIT

> When this flag is given, the entry box will cause the form to stop running if the user pressed return inside of the entry box. This can provide a nice shortcut for users.

After an entry box has been created, its contents can be set by `newtEntrySet()`. After the entry box itself, the new string to place in the entry box is passed. The final parameter, `cursorAtEnd`, controls where the cursor will appear in the entry box. If it is zero, the cursor remains at its present location; a nonzero value moves the cursor to the end of the entry box's new value. While the simplest way to find the value of an entry box is by using a `resultPtr`, doing so complicates some applications. `newtEntryGetValue()` returns a pointer to the string which the entry box currently contains. The returned pointer may not be valid once the user further modifies the entry box, and will not be valid after the entry box has been destroyed, so be sure to save its value in a more permanent location if necessary. Entry boxes allow applications to filter characters as they are entered. This allows programs to ignore characters which are invalid (such as entering a ^ in the middle of a phone number) and provide intelligent aids to the user (such as automatically adding a '.' after the user has typed in the first three numbers in an IP address). When a filter is registered through `newtEntrySetFilter()`, both the filter itself and an arbitrary `void *`, which passed to the filter whenever it is invoked, are

recorded. This data pointer isn't used for any other purpose, and may be `NULL`. Entry filters take four arguments.

1. The entry box which had data entered into it
2. The data pointer which was registered along with the filter
3. The new character which `newt` is considering inserting into the entry box
4. The current cursor position (0 is the leftmost position)

The filter returns 0 if the character should be ignored, or the value of the character which should be inserted into the entry box. Filter functions which want to do complex manipulations of the string should use `newtEntrySet()` to update the entry box and then return 0 to prevent the new character from being inserted. When a callback is attached to a entry box, the callback is invoked whenever the user moves off of the callback and on to another component. Here is a sample program which illustrates the use of both labels and entry boxes.

```
#include <newt.h>
#include <stdlib.h>
#include <stdio.h>

void main(void) {
    newtComponent form, label, entry, button;
    char * entryValue;

    newtInit();
    newtCls();

    newtOpenWindow(10, 5, 40, 8, "Entry and Label Sample");

    label = newtLabel(1, 1, "Enter a string");
    entry = newtEntry(16, 1, "sample", 20, &entryValue,
        NEWT_FLAG_SCROLL | NEWT_FLAG_RETURNEXIT);
    button = newtButton(17, 3, "Ok");
    form = newtForm(NULL, NULL, 0);
    newtFormAddComponents(form, label, entry, button, NULL);

    newtRunForm(form);

    newtFinished();

    printf("Final string was: %s\n", entryValue);

    /* We cannot destroy the form until after we've used the value
       from the entry widget. */
    newtFormDestroy(form);
}
```

4.7. Checkboxes

Most widget sets include checkboxes which toggle between two value (checked or not checked). `Newt` checkboxes are more flexible. When the user presses the space bar on a checkbox, the checkbox's value changes to the next value in an arbitrary sequence (which wraps). Most checkboxes have two items in that sequence, checked or not, but `newt` allows an arbitrary number of value. This is useful when the user must pick from a limited number of choices. Each item in the sequence is a single character, and the sequence itself is represented as a string. The checkbox components displays the character which currently represents its value the left of a text label, and returns the same character as its current value. The default sequence for checkboxes is `" *"`, with `' '` indicating false and `'*'` true.

```
newtComponent newtCheckbox(int left, int top, const char * text, char defV
      const char * seq, char * result);
char newtCheckboxGetValue(newtComponent co);
```

Like most components, the position of the checkbox is the first thing passed to the function that creates one. The next parameter, `text`, is the text which is displayed to the right of the area which is checked. The `defValue` is the initial value for the checkbox, and `seq` is the sequence which the checkbox should go through (`defValue` must be in `seq`. `seq` may be `NULL`, in which case `" *"` is used. The final parameter, `result`, should point to a character which the checkbox should always record its current value in. If `result` is `NULL`, `newtCheckboxGetValue()` must be used to get the current value of the checkbox. `newtCheckboxGetValue()` is straightforward, returning the character in the sequence which indicates the current value of the checkbox If a callback is attached to a checkbox, the callback is invoked whenever the checkbox responds to a user's keystroke. The entry box may respond by taking focus or giving up focus, as well as by changing its current value.

4.8. Radio Buttons

Radio buttons look very similar to checkboxes. The key difference between the two is that radio buttons are grouped into sets, and exactly one radio button in that set may be turned on. If another radio button is selected, the button which was selected is automatically deselected.

```
newtComponent newtRadiobutton(int left, int top, const char * text,
         int isDefault, newtComponent prevButton);
```

```
newtComponent newtRadioGetCurrent(newtComponent setMember);
```

Each radio button is created by calling `newtRadiobutton()`. After the position of the radio button, the text displayed with the button is passed. `isDefault` should be nonzero if the radio button is to be turned on by default. The final parameter, `prevMember` is used to group radio buttons into sets. If `prevMember` is `NULL`, the radio button is assigned to a new set. If the radio button should belong to a preexisting set, `prevMember` must be the previous radio button added to that set. Discovering which radio button in a set is currently selected necessitates `newtRadioGetCurrent()`. It may be passed any radio button in the set you're interested in, and it returns the radio button component currently selected. Here is an example of both checkboxes and radio buttons.

```
#include <newt.h>
#include <stdlib.h>
#include <stdio.h>

void main(void) {
    newtComponent form, checkbox, rb[3], button;
    char cbValue;
    int i;

    newtInit();
    newtCls();

    newtOpenWindow(10, 5, 40, 11, "Checkboxes and Radio buttons");

    checkbox = newtCheckbox(1, 1, "A checkbox", ' ', " *X", &cbValue);

    rb[0] = newtRadiobutton(1, 3, "Choice 1", 1, NULL);
    rb[1] = newtRadiobutton(1, 4, "Choice 2", 0, rb[0]);
    rb[2] = newtRadiobutton(1, 5, "Choice 3", 0, rb[1]);

    button = newtButton(1, 7, "Ok");

    form = newtForm(NULL, NULL, 0);
    newtFormAddComponent(form, checkbox);
    for (i = 0; i < 3; i++)
 newtFormAddComponent(form, rb[i]);
    newtFormAddComponent(form, button);

    newtRunForm(form);
    newtFinished();

    /* We cannot destroy the form until after we've found the current
       radio button */
```

```
    for (i = 0; i < 3; i++)
 if (newtRadioGetCurrent(rb[0]) == rb[i])
     printf("radio button picked: %d\n", i);
    newtFormDestroy(form);

    /* But the checkbox's value is stored locally */
    printf("checkbox value: '%c'\n", cbValue);
}
```

4.9. Scales

It's common for programs to need to display a progress meter on the terminal while it performs some length operation (it behaves like an anesthetic). The scale component is a simple way of doing this. It displays a horizontal bar graph which the application can update as the operation continues.

```
newtComponent newtScale(int left, int top, int width, long long fullValue)
void newtScaleSet(newtComponent co, unsigned long long amount);
```

When the scale is created with `newtScale`, it is given the width of the scale itself as well as the value which means that the scale should be drawn as full. When the position of the scale is set with `newtScaleSet()`, the scale is told the amount of the scale which should be filled in relative to the `fullAmount`. For example, if the application is copying a file, `fullValue` could be the number of bytes in the file, and when the scale is updated `newtScaleSet()` would be passed the number of bytes which have been copied so far.

4.10. Textboxes

Textboxes display a block of text on the terminal, and is appropriate for display large amounts of text.

```
newtComponent newtTextbox(int left, int top, int width, int height, int fl
void newtTextboxSetText(newtComponent co, const char * text);
```

`newtTextbox()` creates a new textbox, but does not fill it with data. The function is passed the location for the textbox on the screen, the width and height of the textbox (in characters), and zero or more of the following flags:

NEWT_FLAG_WRAP

All text in the textbox should be wrapped to fit the width of the textbox. If this flag is not specified, each newline delimited line in the text is truncated if it is too long to fit. When `newt` wraps text, it tries not to break lines on spaces or tabs. Literal newline characters are respected, and may be used to force line breaks.

NEWT_FLAG_SCROLL

The text box should be scrollable. When this option is used, the scrollbar which is added increases the width of the area used by the textbox by 2 characters; that is the textbox is 2 characters wider then the width passed to `newtTextbox()`.

After a textbox has been created, text may be added to it through `newtTextboxSetText()`, which takes only the textbox and the new text as parameters. If the textbox already contained text, that text is replaced by the new text. The textbox makes its own copy of the passed text, so these is no need to keep the original around unless it's convenient.

4.10.1. Reflowing Text

When applications need to display large amounts of text, it's common not to know exactly where the linebreaks should go. While textboxes are quite willing to scroll the text, the programmer still must know what width the text will look "best" at (where "best" means most exactly rectangular; no lines much shorter or much longer then the rest). This common is especially prevalent in internationalized programs, which need to make a wide variety of message string look god on a screen. To help with this, `newt` provides routines to reformat text to look good. It tries different widths to figure out which one will look "best" to the user. As these commons are almost always used to format text for textbox components, `newt` makes it easy to construct a textbox with reflowed text.

```
char * newtReflowText(char * text, int width, int flexDown, int flexUp,
        int * actualWidth, int * actualHeight);
newtComponent newtTextboxReflowed(int left, int top, char * text, int widt
      int flexDown, int flexUp, int flags);
int newtTextboxGetNumLines(newtComponent co);
```

`newtReflowText()` reflows the `text` to a target width of `width`. The actual width of the longest line in the returned string is between `width - flexDown` and `width + flexUp`; the actual maximum line length is chosen to make the displayed

check look rectangular. The `int`s pointed to by `actualWidth` and `actualHeight` are set to the width of the longest line and the number of lines in in the returned text, respectively. Either one may be `NULL`. The return value points to the reflowed text, and is allocated through `malloc()`. When the reflowed text is being placed in a textbox it may be easier to use `newtTextboxReflowed()`, which creates a textbox, reflows the text, and places the reflowed text in the listbox. It's parameters consist of the position of the final textbox, the width and flex values for the text (which are identical to the parameters passed to `newtReflowText()`, and the flags for the textbox (which are the same as the flags for `newtTextbox()`. This function does not let you limit the height of the textbox, however, making limiting it's use to constructing textboxes which don't need to scroll. To find out how tall the textbox created by `newtTextboxReflowed()` is, use `newtTextboxGetNumLines()`, which returns the number of lines in the textbox. For textboxes created by `newtTextboxReflowed()`, this is always the same as the height of the textbox. Here's a simple program which uses a textbox to display a message.

```
#include <newt.h>
#include <stdlib.h>

char message[] = "This is a pretty long message. It will be displayed "
   "in a newt textbox, and illustrates how to construct "
   "a textbox from arbitrary text which may not have "
   "very good line breaks.\n\n"
   "Notice how literal \\n characters are respected, and "
   "may be used to force line breaks and blank lines.";

void main(void) {
    newtComponent form, text, button;

    newtInit();
    newtCls();

    text = newtTextboxReflowed(1, 1, message, 30, 5, 5, 0);
    button = newtButton(12, newtTextboxGetNumLines(text) + 2, "Ok");

    newtOpenWindow(10, 5, 37,
     newtTextboxGetNumLines(text) + 7, "Textboxes");

    form = newtForm(NULL, NULL, 0);
    newtFormAddComponents(form, text, button, NULL);

    newtRunForm(form);
    newtFormDestroy(form);
    newtFinished();
}
```

4.11. Scrollbars

Scrollbars (which, currently, are always vertical in `newt`), may be attached to forms to let them contain more data then they have space for. While the actual process of making scrolling forms is discussed at the end of this section, we'll go ahead and introduce scrollbars now so you'll be ready.

```
newtComponent newtVerticalScrollbar(int left, int top, int height,
        int normalColorset, int thumbColorset);
```

When a scrollbar is created, it is given a position on the screen, a height, and two colors. The first color is the color used for drawing the scrollbar, and the second color is used for drawing the thumb. This is the only place in newt where an application specifically sets colors for a component. It's done here to let the colors a scrollbar use match the colors of the component the scrollbar is mated too. When a scrollbar is being used with a form, `normalColorset` is often `NEWT_COLORSET_WINDOW` and `thumbColorset` `NEWT_COLORSET_ACTCHECKBOX`. Of course, feel free to peruse `<newt.h>` and pick your own colors. As the scrollbar is normally updated by the component it is mated with, there is no public interface for moving the thumb.

4.12. Listboxes

Listboxes are the most complicated components `newt` provides. They can allow a single selection or multiple selection, and are easy to update. Unfortunately, their API is also the least consistent of `newt`'s components. Each entry in a listbox is a ordered pair of the text which should be displayed for that item and a *key*, which is a `void *` that uniquely identifies that listbox item. Many applications pass integers in as keys, but using arbitrary pointers makes many applications significantly easier to code.

4.12.1. Basic Listboxes

Let's start off by looking at the most important listbox functions.

```
newtComponent newtListbox(int left, int top, int height, int flags);
int newtListboxAppendEntry(newtComponent co, const char * text,
      const void * data);
void * newtListboxGetCurrent(newtComponent co);
void newtListboxSetWidth(newtComponent co, int width);
```

```
void newtListboxSetCurrent(newtComponent co, int num);
void newtListboxSetCurrentByKey(newtComponent co, void * key);
```

A listbox is created at a certain position and a given height. The `height` is used for two things. First of all, it is the minimum height the listbox will use. If there are less items in the listbox then the height, suggests the listbox will still take up that minimum amount of space. Secondly, if the listbox is set to be scrollable (by setting the `NEWT_FLAG_SCROLL flag`, the `height` is also the maximum height of the listbox. If the listbox may not scroll, it increases its height to display all of its items. The following flags may be used when creating a listbox:

NEWT_FLAG_SCROLL

The listbox should scroll to display all of the items it contains.

NEWT_FLAG_RETURNEXIT

When the user presses return on an item in the list, the form should return.

NEWT_FLAG_BORDER

A frame is drawn around the listbox, which can make it easier to see which listbox has the focus when a form contains multiple listboxes.

NEWT_FLAG_MULTIPLE

By default, a listbox only lets the user select one item in the list at a time. When this flag is specified, they may select multiple items from the list.

Once a listbox has been created, items are added to it by invoking `newtListboxAppendEntry()`, which adds new items to the end of the list. In addition to the listbox component, `newtListboxAppendEntry()` needs both elements of the (text, key) ordered pair. For lists which only allow a single selection, `newtListboxGetCurrent()` should be used to find out which listbox item is currently selected. It returns the key of the currently selected item. Normally, a listbox is as wide as its widest element, plus space for a scrollbar if the listbox is supposed to have one. To make the listbox any larger then that, use `newtListboxSetWidth()`, which overrides the natural list of the listbox. Once the width has been set, it's fixed. The listbox will no longer grow to accommodate new entries, so bad things may happen! An application can change the current position of the listbox (where the selection bar is displayed) by calling `newtListboxSetCurrent()` or `newtListboxSetCurrentByKey()`. The first sets the current position to the entry number which is passed as the second

argument, with 0 indicating the first entry. `newtListboxSetCurrentByKey()` sets the current position to the entry whose `key` is passed into the function.

4.12.2. Manipulating Listbox Contents

While the contents of many listboxes never need to change, some applications need to change the contents of listboxes regularly. `Newt` includes complete support for updating listboxes. These new functions are in addition to `newtListboxAppendEntry()`, which was already discussed.

```
void newtListboxSetEntry(newtComponent co, void * key, const char * text);
int newtListboxInsertEntry(newtComponent co, const char * text,
                           const void * data, void * key);
int newtListboxDeleteEntry(newtComponent co, void * key);
void newtListboxClear(newtComponent co);
```

The first of these, `newtListboxSetEntry()`, updates the text for a key which is already in the listbox. The `key` specifies which listbox entry should be modified, and `text` becomes the new text for that entry in the listbox. `newtListboxInsertEntry()` inserts a new listbox entry *after* an already existing entry, which is specified by the `key` parameter. The `text` and `data` parameters specify the new entry which should be added. Already-existing entries are removed from a listbox with `newtListboxDeleteEntry()`. It removes the listbox entry with the specified `key`. If you want to remove all of the entries from a listbox, use `newtListboxClear()`.

4.12.3. Multiple Selections

When a listbox is created with `NEWT_FLAG_MULTIPLE`, the user can select multiple items from the list. When this option is used, a different set of functions must be used to manipulate the listbox selection.

```
void newtListboxClearSelection(newtComponent co);
void **newtListboxGetSelection(newtComponent co, int *numitems);
void newtListboxSelectItem(newtComponent co, const void * key,
                    enum newtFlagsSense sense);
```

The simplest of these is `newtListboxClearSelection()`, which deselects all of the items in the list (listboxes which allow multiple selections also allow zero selections). `newtListboxGetSelection()` returns a pointer to an array which contains the keys for all of the items in the listbox currently selected. The `int` pointed to by `numitems` is set to the number of items currently selected (and hence

the number of items in the returned array). The returned array is dynamically allocated, and must be released through `free()`. `newtListboxSelectItem()` lets the program select and deselect specific listbox entries. The `key` of the listbox entry is being affected is passed, and `sense` is one of `NEWT_FLAGS_RESET`, which deselects the entry, `NEWT_FLAGS_SET`, which selects the entry, or `NEWT_FLAGS_TOGGLE`, which reverses the current selection status.

4.13. Advanced Forms

Forms, which tie components together, are quite important in the world of `newt`. While we've already discussed the basics of forms, we've omitted many of the details.

4.13.1. Exiting From Forms

Forms return control to the application for a number of reasons:

- A component can force the form to exit. Buttons do this whenever they are pushed, and other components exit when `NEWT_FLAG_RETURNEXIT` has been specified.
- Applications can setup hot keys which cause the form to exit when they are pressed.
- `Newt` can exit when file descriptors are ready to be read or ready to be written to.

By default, `newt` forms exit when the F12 key is pressed (F12 is setup as a hot key by default). `Newt` applications should treat F12 as an "Ok" button. If applications don't want F12 to exit the form, they can specify `NEWT_FLAG_NOF12` as flag when creating the form with `newtForm`.

```
void newtFormAddHotKey(newtComponent co, int key);
void newtFormWatchFd(newtComponent form, int fd, int fdFlags);

void newtDrawForm(newtComponent form);
newtComponent newtFormGetCurrent(newtComponent co);
void newtFormSetCurrent(newtComponent co, newtComponent subco);
void newtFormRun(newtComponent co, struct newtExitStruct * es);

newtComponent newtForm(newtComponent vertBar, const char * help, int flags
void newtFormSetBackground(newtComponent co, int color);
void newtFormSetHeight(newtComponent co, int height);
void newtFormSetWidth(newtComponent co, int width);
```