TTK, An iPod GUI Library

Joshua Oreman

Sunday, December 1, 2005

Contents

Chapter 1

Architecture

TTK is a library for the creation of graphical user interfaces on the iPod, along with a few other things.

This chapter will present all the information you need to program the library, including recommended ways to do various things, *except* the actual function calls and data structures themselves. Those are documented in later chapeters. Functions may be mentioned, but only when understanding them is essential to understanding the library as a whole.

1.1 Why Use TTK?

"Wait!" I can hear you wondering. "Why should I use TTK? I've already learned Nano-X/SDL/whatever, why should I switch now?" Well, that's an objection that's inevitably going to come up, and I hope to answer it here. The first thing I'd recommend is to *read* this introductory chapter; it describes most of what TTK is about. But if you need some quick reasons, here they are:

- TTK is a higher-level library than what you're used to, so you can do larger things more easily and not have to reinvent the wheel quite so often.

- TTK has intelligent event handling, with things like held-button timers; scrollwheel taps; making sure different parts of a button press and release aren't sent to different widgets; turning a bunch of "button press `r`" events into one, more efficient "scroll right by x"; etc.
- TTK only draws your stuff when it needs to be drawn, and sleeps rather a lot during its event loop; thus, background processes like MPD may well run more quickly.
- TTK acts as an abstraction layer between your code and a lower-level graphics library; if someone develops FooGraphics for the iPod, and it runs ten times as fast as current code, making TTK programs use it is a simple matter of writing a backend for TTK.
- TTK makes easy things easy. It takes about ten lines of code (not including the menu definitions) to make a program that displays a menu, with submenus.
- TTK makes some hard things easy. Want to save your settings on exit from that menu? Fine, just set `menu->down = my_down;` and make `my_down` save settings if the button pressed is MENU, before calling `ttk_menu_down`.
- TTK makes hard things possible. You have a lot of flexibility, and you can do pretty low-level things if you really need to. Heck, you can blast data right to the LCD screen if you want to do that.
- TTK is extensible. New widgets that fit in with the existing ones are quite easy to write (comparatively for a GUI), and people can use your code by calling `new_my_widget` just like they might call `ttk_new_menu_widget`.

1.2 Windows

An application written with TTK will be comprised of multiple "windows", each usually opening from a menu (itself a window, etc). For instance, the main menu of `podzilla` is a window, as is the submenu opened when you select "Extras", as is the window opened when you select "Hunt the Wumpus."

This stacked-window paradigm is different from the one seen on most desktop systems: you can't drag windows around, there is no "window dressing", and you aren't really supposed to have multiple windows per application (though it certainly can be done).

Basically, windows function as a stack, of which usually only the topmost item is visible; when a popup window like an alert or dialog is on the stack, you can still see the appropriate parts of the obscured window, but it receives no events until the upper window is closed. Basically, the "window stack" functions as more of a useful bookkeeping device than anything, allowing e.g. any widget to easily move up a window, whatever is in said window, and letting you "minimize" windows (send temporarily to bottom of stack), change their Z-axis order, etc.

TTK windows aren't really like pure X11 windows either; the X11 windows are just canvasses, while TTK windows have some important associated information. For instance, each window can define a title, which will be shown in the header, assuming you haven't set said header to be hidden (another window property). Basically, the properties of windows have been morphed into those useful for an iPod user interface—one with a lot of hierarchial GUI elements. Just remember that a window contains, generally, everything you want on the screen for the user to interact with at once, and you'll be fine.

1.3 Widgets

But windows aren't all; if they were, you'd have an app consisting of a lot of titled, blank canvasses.

Windows contain zero or more *widgets*, which are just **GUI elements that can be drawn on the screen and possibly take input from the user**. Widgets can be either *output widgets* or *input widgets*; don't let the names confuse you, the latter is a superset of the former. Output widgets take no input; their output can be variable, but it depends on external factors (like the state of another widget or the charge of a battery). Input widgets are the ones that form the backbone of the system. When focused, they capture all input and can perform widget-dependent actions based on it. Examples might include a menu, text box, file editor, hypertext browser, etc. There

can only be one top-level input widget per window. If you want more, make a widget that manages that! `:-)` Input widgets are identified by setting the focusable flag in the widget structure.

Sometimes we will refer to a window's "focused widget." That's just the first input widget of that window. This means you *can* have more than one input widget per window; the others will simply be displayed, but will receive no events. Thus, if you want to be able to switch between widgets (e.g. in a form), you need to make *one* widget that will be put in the window and have all other input widgets inserted into it; that widget can figure out a good way to move input between its children. (Note that TTK does not actually have any concept of widget or window "parents" or "children"; the word is used here simply in an illustrative context.)

Widget structures contain, along with various settings like positioning and size, several function pointers. These are scroll, stap, button, down, held, frame, timer, and destroy. Each is called when the appropriate event happens. (The events will be described in detail below.) Each event handler is expected to do whatever it needs with the widget structure, determine whether the widget needs to be redrawn, and if so set the *dirty* flag.

The dirty flag is used to save CPU cycles and make TTK applications faster. Basically, widgets will not be drawn on the screen (actually, the window's double buffer) unless the flag is set. It is cleared after each draw. Therefore, every time you make a change to a widget and expect it to be seen, you **must set the dirty flag**. If widgets are not being drawn, that's the first thing to check.

Finally, each widget has a widget-specific data member (a `void *`) that can be whatever you want. Most complex widgets will use it to store a pointer to a structure containing widget-specific parameters; some simple widgets might not need to use it at all. TTK itself couldn't care less about the value of this parameter; it is only intended for use by widget handler functions.

1.4 Events

Any time something interesting happens in TTK, it creates an *event*. There are three main types of events: immediate events, periodic events, and computed events.

1.4.1 Immediate Events

Immediate events are those created immediately by some input on the iPod (or your keyboard for a desktop build); they are the down (button press), button (button release), and scroll (scrollwheel movement) events. These events will be passed first to a *global event handler*, which can be set by the application. That event handler may do whatever it wants, and returns a code indicating whether or not the event should be processed further. If not, it is thrown out, and the event will never make it to the widget.

scroll events are generated, by default, 96 to a full rotation of the wheel. This is quite a high speed, and it is certainly not expected that one "step" will move one item; instead, each widget is intended to put a line at the beginning of their scroll handler: `TTK_SCROLLMOD(dir,`N`)`. This will modulate the scrolling by N; for instance, if $N = 5$, the widget will process 19–20 scroll events per rotation. (On the iPod, the `SCROLLMOD` macro simply returns 0 from the handler if the event is not to be processed; it uses a static variable called `sofar`, which it creates itself. On the desktop, it is ignored.) You can, of course, use another mechanism for processing scroll events; the slider widget modulates them itself so that one rotation covers the distance between the minimum and maximum values on the slider. The global *wheel sensitivity* is independent of this scroll modulation.

Furthermore, you can use the `TTK_SCROLL_ACCEL` macro to emulate the iPod's wheel acceleration feature. This allows for both coarse and fine adjustment of long lists. Put this in your scroll handler as well. Unless you want to allow for very fine adjustments, it should be used in conjunction with `TTK_SCROLLMOD`. The syntax is `TTK_SCROLL_ACCEL(dir`, *slow*, *max*`)`, where dir is the scroll event to be amplified, *slow* is a factor inversely proportional to the rate of acceleration, and *max* is the maximum acceleration factor. For example, `TTK_SCROLL_ACCEL(event->arg, 5, 20)` will accelerate every fifth Podzilla2 scroll event, up to a maximum factor of 20.

input is also an immediate event, but it is rather different from the rest; see "Text Input" below.

After immediate events get past the global event handler, they are dispatched to the focused widget of the top window. There are, however, a few situations that cause immediate events *not* to fire:

- If a key is repeating, different parts of the same repeat (down events without intervening button events) will not be sent to different widgets.
- A down event and the corresponding button event will not be sent to different widgets.
- Pressing a button will cancel any stap event that might have been fired; see below.

1.4.2 Periodic Events

Periodic events are events that fire every certain amount of time; they are the frame and timer events. They have separate handlers, but they differ only in the way they are set: frame events are set with a frames-per-second value, which is used to compute the milliseconds of delay, while timer events are set with a direct milliseconds value. Thus, one could use them for two separate but equally important timers; or one could use e.g. frame for a game's loop and timer for a one-shot timer (the handler would have to deactivate the timer if you wanted one-shot behavior).

1.4.3 Computed Events

Computed events are those that are fired based on information gleaned through immediate events and through (on an iPod) examining hardware registers. There are two: held and stap.

held events are fired whenever a button is held down for more than the holdtime value (in milliseconds) specified in the focused widget. The button does not have to be released for the event to be fired.

stap events are fired, on 4G iPods and above, when the user taps the scroll wheel. Various heuristics are used to differentiate between stap events, scroll events, and down events. They generally work very well, though one of their consequences is that scrollwheel tap events cannot be detected until the user lifts his/her finger from the wheel surface. This is probably not optimal behavior, but there is no other way to be sure of a user's intent; he may, for instance, be lingering before starting to scroll. Here are the heuristics, for those who are interested:

- As stated above, pushing a button cancels any otherwise-good scroll-wheel tap.
- If the user's finger moved more than 1/20 of the wheel's circumference while touching it, the stap event is thrown out.
- If the user held his/her finger for more than 2/5 of a second, the event is thrown out.
- Otherwise, a stap event will be fired.

1.5 Text Input

TTK has a generic mechanism to let you create *text input methods* that allow the user to input text using only the interface of the iPod. The specific mechanism for text input (be it an onscreen keyboard, telephone keypad, whatever) is not specified.

Basically, a text input method is just a widget written in a special way. Text input is started with `ttk_input_start(method)` from a widget. From that point on, the widget will not receive any immediate or computed events. Instead, these will go to the text input method widget. When the user has fully specified a character, the text input method calls `ttk_input_char(ch)`, which causes an input event to be delivered to the widget that originally started text input. To end text input, either the text input method or the calling widget may call `ttk_input_end()`, which also frees the input method structure.

These functions are intended to be called on a window that is already shown onscreen. If you want to prepare text input for a window that isn't shown yet, you can use the functions like `ttk_input_start_for(win,meth)` that let you specify the window.

Different text input methods vary in the amount of screen space they need. Something like a Morse code input might need only a single status line at the bottom, whereas an on-screen keyboard might need much more space. As such, `ttk_input_start()` will return the *height* of the text input method's screen space, so that the calling widget can adjust its own size. Some input methods, like a telephone keypad, might need a square patch of

space instead of a line all the way across the bottom of the screen; these can be positioned in a convenient location by `ttk_input_move()` for fullscreen applications.

1.6 Graphics Drawing

TTK supports two backends at present: SDL and Nano-X. As such, to support users coming from a background of either, it has two main graphics-drawing interfaces.

For both interfaces, the core types are the same. A `ttk_surface` represents an area where things can be drawn: basically, a rectangular array of pixels. A `ttk_color` represents the color of a pixel; it would differ depending on whether the program was run on a 24-bpp desktop, the iPod Photo, a monochrome iPod, etc. A `ttk_point` represents a point, with `x` and `y` members. Finally, a `ttk_font` represents a font.

The SDL-like graphics drawing interface is that exported by funtions with names like `ttk_rect`, `ttk_fillellipse`, and `ttk_text`. The Nano-X-like functions have a `_gc` suffix: `ttk_rect_gc`, `ttk_fillellipse_gc`, `ttk_text_gc`, etc. The SDL-like functions pass parameters like fonts and colors as arguments to the function, while the Nano-X-like ones use a **graphics context**. A graphics context stores five things: a foreground color, a background color (used for text drawing only), a flag indicating whether to use that background color, a font, and a flag indicating whether to use `ttk_fillrect` as a region inverter instead of a region filler (the `xormode` flag). Graphics context support is intended for the easier migration of Nano-X applications and Nano-X users.

The graphics drawing functions themselves work as one would expect, and will not be documented here. See Chapter 3 for the full details regarding which functions are available.

1.7 Appearance

In order to give the user more flexibility in choosing how they want their applications to look, TTK supports an *appearance* facility. This loads specially formatted text files (see Chapter 5 for the format), each consisting of a

bunch of named entries that can have three properties: a background *image*, a background *color*, and some *spacing*. The spacing is used to shrink or grow rectangles so they have some padding around them; or to move vertical lines left or right, or horizontal lines up or down.

In order to support this useful facility, there is not that much you need to do. Mainly, replace calls like these:

```
ttk_rect(srf, x, y, w, h, color);
```

with calls like these.

```
ttk_ap_rect(srf, ttk_ap_get("property"), x, y, w, h);
```

The same applies to `fillrect`s, and horizontal and vertical `line`s. The `ttk_ap_*` calls ensure that all three properties—color, image, and spacing—are supported where applicable. However, you may want just the color, for instance to figure out what color to make some text. In that case, you should use something like

```
ttk_text(srf, font, x, y, ttk_ap_getx("property")->color, str);
```

Note the use of `ttk_ap_getx`; this causes the returned value on not finding an entry to be a pointer to a structure with dummy values, instead of a NULL pointer. You should always use `ttk_ap_getx` when you will be immediately dereferencing the result, so as to avoid a segmentation fault.

The dummy values are, by the way: no spacing, no image, **black** color, and a name of `"NO_SUCH_ITEM"`.

1.8 Nano-X Emulation

The main player in the iPod graphics field before TTK was Nano-X (formerly Microwindows). Since a rather large body of applications was written for Nano-X, not the least of which was `podzilla`, an API emulation layer was necessary to allow easy and gradual porting. One of the reasons that an earlier iPod GUI library, PTK, failed to make much of an impact was its

incompatibility with what had come before it; TTK does not intend to fall into such a situation. Thus, the emulation.

The Nano-X emulation encompasses 45 functions, enough for full Podzilla support. Notable unimplemented functions include `GrSetWindowBackgroundColor`, `GrSetScreenSaverTimeout`, and `GrArc`. All implemented functions have a `t_` at the beginning of their names, e.g. `t_GrFillRect`; if you do not `#define MWBACKEND`, the non-`t_` versions will be `#define`d to their `t_` equivalents.

This emulation has met with great success in, and has in part been shaped by, the porting of `podzilla` to TTK. That is: **It works**.

Chapter 2

Macros and Globals

2.1 Macros

```
#define TTK_POD_X11       0
#define TTK_POD_1G        01
#define TTK_POD_2G        02
#define TTK_POD_3G        04
#define TTK_POD_PP5002    07
#define TTK_POD_4G        010
#define TTK_POD_MINI_1G   020
#define TTK_POD_PHOTO     040
#define TTK_POD_PP5020    070
#define TTK_POD_MINI_2G   0100
#define TTK_POD_PP5022    0700
#define TTK_POD_PP502X    0770
```

These are the bitmasks for possible return values from `ttk_get_podversion()`.

```
#define TTK_BUTTON_ACTION   '\n'
#define TTK_BUTTON_PREVIOUS 'w'
#define TTK_BUTTON_NEXT     'f'
#define TTK_BUTTON_MENU     'm'
#define TTK_BUTTON_PLAY     'd'
#define TTK_BUTTON_HOLD     'h'
```

These are the definitions of buttons; whenever you see a parameter called `button`, it will be one of these.

```
#define TTK_MOVE_ABS      0
#define TTK_MOVE_REL      1
#define TTK_MOVE_END      2
```

These are the arguments to the `ttk_move_window()` function.

```
#define TTK_DIRTY_HEADER     1
#define TTK_DIRTY_WINDOWAREA 2
#define TTK_FILTHY           3
extern int ttk_dirty;
```

These control what big things need to be redrawn right now. If you do `ttk_dirty |= TTK_DIRTY_HEADER`, the header will be redrawn. `TTK_DIRTY_WINDOWAREA` will redraw the current window on the screen, but *not* its widgets. (They'll be at their state when they were last drawn.) `TTK_DIRTY_FILTHY` does both of these, and possibly more. Basically, it redraws everything.

```
#define TTK_EV_CLICK   1
#define TTK_EV_UNUSED  2
#define TTK_EV_DONE    4
#define TTK_EV_RET(x)  ((x)<<8)
```

These are the possible return values (bitwise OR) from event handlers, with the exception of the global event handler (its return value indicates whether or not to process the event further).

```
#define TTK_INPUT_ENTER  '\r'
#define TTK_INPUT_BKSP   '\b'
#define TTK_INPUT_LEFT   '\1'
#define TTK_INPUT_RIGHT  '\2'
#define TTK_INPUT_END    '\0'
```

These are the special characters that can be passed to an input event; most of them are self-explanatory. `TTK_TEXT_END` is sent just *after* `ttk_input_end()` is called; this gives you a chance to e.g. show a menu or something.

`TTK_EV_CLICK` causes TTK to emit a clicking sound. See `ttk_click()`.

`TTK_EV_UNUSED` causes TTK to call the global unused handler for this event, if one is defined.

`TTK_EV_DONE` causes `ttk_run()` to return after processing this event.

`TTK_EV_RET(x)` in conjunction with `TTK_EV_DONE` causes `ttk_run()`'s return value to be `x`.

```
#define  WHITE   255, 255, 255
#define   GREY   160, 160, 160
#define DKGREY    80,  80,  80
#define  BLACK     0,   0,   0
```

These are the allowable colors on black-and-white iPods. They should be used in a context like

```
mycolor = ttk_makecol (WHITE);
```

2.2 Globals

```
extern TWindowStack *ttk_windows;
extern ttk_font ttk_menufont, ttk_textfont;
extern ttk_screeninfo *ttk_screen;
extern ttk_fontinfo *ttk_fonts;
extern int ttk_epoch;
```

These are TTK's globals. See below for information about their structures.

`ttk_windows` is the stack of all windows onscreen, including those that are invisible because they are obscured. The first item is the top window.

`ttk_menufont` is the font used for "menu-like" things: the header; menu widgets; larger text; etc. It must be defined right after you initialize TTK.

`ttk_textfont` is the font used for "text-like" things: textarea widgets, etc. It probably shouldn't be bigger than `ttk_menufont`, and also must be defined early on.

`ttk_screen` contains information about the current screen setup, as well as the screen surface itself. The most important members are `ttk_screen->w`, `ttk_screen->h`, and `ttk_screen->bpp`.

`ttk_fonts` is a linked list of fonts, along with information about each one. You probably shouldn't traverse this by hand, but you can if you want to.

`ttk_epoch` is an odd little variable that deserves special mention. Basically, it's a "redraw everything" mechanism. Anything that needs to draw stuff is supposed to keep its own private `epoch` variable, and check it against `ttk_epoch` as often as possible. If there is a mismatch, everything is redrawn. This applies not only to the obvious—`ttk_run()` controlling the drawing of widgets—but also to things like the menu widget, textarea widget, etc. that make double-buffer surfaces when initialized so they can draw themselves more quickly.

The end result of all this is that when you change the font or the color scheme, you should increment `ttk_epoch`.

Chapter 3

Data Structures

The TTK API contains several important data structures. In general, the "major" structures (appearance stuff, widgets, windows, and their ilk) are named `TSomething`, while the "minor" ones (screen info, font info, points, etc.) are named `ttk_something`. The structures are described below.

3.1 TWindow

```
typedef struct TWindow
{
    const char *title;
    TWidgetList *widgets;
    int x, y, w, h, color;
    ttk_surface srf;
    int titlefree;
    int dirty;
    struct TApItem *background;
    /* readonly */ struct TWidget *focus;

    int data;
} TWindow;
```

`title` stores the window's title for the header.

`widgets` is a list of all widgets in the window.

`x, y, w, h` indicates where the window goes onscreen. In general, you'll have `x = ttk_screen->wx` (usually 0), `y = ttk_screen->wy` (about 20), and `w` and `h` filling up the rest of the screen. If you change these, that makes it a popup window, and you'll get a border around it.

`color` is a *flag* that indicates whether or not this window supports color. *It has nothing to do with a particular color.*

`srf` is the window's double buffer. All widgets in the window will draw here, and then it will be blitted to the screen every so often. You needn't concern yourself with the details.

`titlefree` is a flag that indicates whether `title` should be freed when the window is freed. It is set by `ttk_window_set_title()`.

`dirty` is a dirty flag for the whole window. If you set this, *all* widgets in the window will be redrawn.

`background` is an appearance item for the background of the window. If you leave this set to its default of 0, `ttk_ap_getx("window.bg")` will be used instead.

`focus` points to the widget in `widgets` that has focus. You shouldn't modify this yourself, but you can use it to determine what the focused widget is.

`data` is an `int` that you can use for whatever you want.

3.2 TWidget

```
typedef struct TWidget
{
    int x, y, w, h;
    int focusable;
    int dirty;
    int holdtime;
    int keyrepeat;
```

```
    int rawkeys;
    /* readonly */ TWindow *win;

    void (*draw) (struct TWidget *this, ttk_surface srf);
    int (*scroll) (struct TWidget *this, int dir);
    int (*stap) (struct TWidget *this, int where);
    int (*button) (struct TWidget *this, int button, int time);
    int (*down) (struct TWidget *this, int button);
    int (*held) (struct TWidget *this, int button);
    int (*input) (struct TWidget *this, int ch);
    int (*frame) (struct TWidget *this);
    int (*timer) (struct TWidget *this);
    void (*destroy) (struct TWidget *this);

    void *data;
} TWidget;
```

`x, y, w, h` indicate the geometry of the widget. A rectangle of this position and size will be cleared prior to calling draw. If you don't want any clearing, set all four to zero.

`focusable` must be set to 1 if you want the widget to receive any events other than frame and timer.

`dirty` must be set to 1 or above when you want the widget to be redrawn. It is set to 0 every time draw is called.

`holdtime` is the amount of time, in milliseconds, that a button must be held down for a held event to fire. The default is one second, 1000 ms.

`keyrepeat` must be set to 1 if you want to receive key-repeat events (they will be received as multiple downs for one button.

`rawkeys` controls the events passed to this widget. If it is set to 0 (the default), they will be passed as described in the rest of this manual. If set to 1, however, no scroll events will be passed, and down and button will receive ASCII keys as arguments. This is intended for use in text-input methods for a full-size keyboard.

`win` stores a pointer to the window this widget is in. This is set by `ttk_add_widget()` and should not be modified, or used before that function is called.

draw points to a function that will be called whenever the widget needs to be drawn. The first argument points to the widget; the second is the surface it should draw itself on.

scroll points to the event handler for scroll events. The first argument, as usual, is the widget, while the second indicates how far the user scrolled. (The units of "how far" are 96 to a full rotation.) It should return a normal event code; see the definitions of `TTK_EV_*` above.

stap points to the event handler for scrollwheel tap events; the second argument indicates where on the scrollwheel the user tapped. 0 is at the top, on top of the Menu button; numbers increase as you move clockwise around the wheel, and 95 is directly to the left of 0. Returns a normal event code.

button points to the event handler for button release events. The second argument indicates which button was released, and the third tells how long it was pressed for, in milliseconds. Returns a normal event code.

down points to the event handler for button press events. The second argument shows which button was pressed. Returns a normal event code.

held is the event handler for button held events. The second argument refers to which button was held. It should return a normal event code.

input is called whenever a character is input after `ttk_input_start()` has been called. During this time, no other events will be passed to the widget until `ttk_input_end()`.

frame **and** timer are the two periodic event handlers discussed in Chapter 1. They should return `TTK_EV_DONE` if `ttk_run()` should return, and 0 otherwise.

destroy should free any widget-specific data in `data`, including `data` itself. It should not free the widget.

`data` can be whatever the widget wishes to use it for.

3.3 TWidgetList and TWindowStack

Bunches of widgets and windows are managed with two similar linked-list structures.

```
typedef struct TWindowStack {
    TWindow *w;
    int minimized;
    struct TWindowStack *next;
} TWindowStack;

typedef struct TWidgetList
{
    struct TWidget *v;
    struct TWidgetList *next;
} TWidgetList;
```

In both structures, `next` points to the next entry in the linked list, or `NULL` if this is the last entry. `v` or `w` points to the actual item; they are made different in name so it is more difficult to confuse the two structures. Finally, in `TWindowStack` only, the `minimized` item indicates that the window should be moved down in the stack whenever it reaches the top.

3.4 TApItem

This structure defines the "appearance item", which can be fetched with `ttk_ap_get[x]()` and used with the `ttk_ap_*` functions.

```
#define TTK_AP_COLOR   1
#define TTK_AP_IMAGE   2
#define TTK_AP_SPACING 4

typedef struct TApItem
{
    char *name;
    int type;
```

```
    ttk_color color;
    ttk_surface img;
    int spacing;
    struct TApItem *next;
} TApItem;
```

`name` is the name of the property that was fetched, or `"NO_SUCH_ITEM"` if it was the result of a failed search.

`type` is a bitwise OR of the `TTK_AP_*` constants, indicating which of those properties are defined for this key. Those that are not defined are liable to contain garbage.

`color` specifies the color for this item if `type & TTK_AP_COLOR`.

`img` specifies the background image for this item if `type & TTK_AP_IMAGE`.

`spacing` specifies the spacing value for this item if `type & TTK_AP_SPACING`.

`next` is a linked-list pointer, used internally.

Note that the `TApItem`s you get from `ttk_ap_get()` and friends are pointers to internal data structures and should not be modified or freed.

3.5 `ttk_fontinfo` and `ttk_screeninfo`

```
typedef struct ttk_fontinfo {
    char file[64];
    char name[64];
    int size;
    ttk_font f;
    int loaded;
    int good;
    int offset;
    struct ttk_fontinfo *next;
} ttk_fontinfo;
```

Fonts are managed by a `ttk_fontinfo` structure that looks like this. These structures generally occur in the linked list of `ttk_fonts` (a global), and you can get the closest match for a particular font with the function `ttk_get_fontinfo()`. (If you just want to draw text with the font, you don't need this structure; it's only for those writing something like a font selector.)

`file` is the filename of the font, including extension, relative to the font path on this system. On the iPod, the font path is `/usr/share/fonts`; for desktop builds, it's `fonts/` in the current directory.

`name` is the human-readable name of the font, which may include spaces.

`size` is the size of the font in pixels, as specified in the font list file. This may or may not be the true size of the font, depending on how much you trust the user.

`f` is the font itself. Unless `loaded` and `good` are both true, this points to garbage.

`loaded` is a flag indicating whether the font has been loaded from disk. Loading fonts takes a long time on the iPod—a second or two per font—and those with lots of fonts would certainly not appreciate a one-minute delay on program startup. Thus, fonts are loaded only on demand (when you call `ttk_get_font()` or `ttk_get_fontinfo()`).

`good` is a flag indicating whether the font has been *successfully* loaded from disk.

`offset` is an offset in vertical pixels to be used in drawing the font. Some fonts have their characters too high in the bounding box; this lowers them to a good position. The value is completely up to whoever makes the font list file.

`next` is a pointer to the next `ttk_fontinfo` structure, if this one is in a linked list.

```
typedef struct ttk_screeninfo {
    int w, h, bpp;
```

```
    int wx, wy;
    ttk_surface srf;
} ttk_screeninfo;
```

Finally, the `ttk_screeninfo` structure gives information about the current screen setup. `w`, `h`, and `bpp` give the screen's width, height, and bits per pixel; `bpp` will always be either 2 or 16. `wx` and `wy` give the (x, y) coordinates of the upper-left hand corner of fullscreen windows; this allows a runtime-specific header height. Finally, `srf` is the screen surface itself, which you shouldn't mess with unless you know what you're doing.

Chapter 4

Function Reference

This chapter contains a reference to all the functions in TTK, sorted into categories.

4.1 Initialization and Querying

```
TWindow *ttk_init();
int ttk_run();
void ttk_quit();
int ttk_get_podversion();
void ttk_get_screensize (int *w, int *h, int *bpp);
void ttk_set_emulation (int w, int h, int bpp);
```

These functions are used to initialize the library and ask various things of it.

`ttk_init` initializes the library. This should be called as one of the first things your program does; you can't use any other functions until you call it, unless otherwise indicated. **Returns** an initial window, already initialized and shown for you. If you don't want this window for some reason, feel free to hide and free it.

`ttk_quit` uninitializes the library. This should be called before your program exits, as you otherwise probably won't get your original terminal restored. You might want to set this as an `atexit()` handler.

`ttk_get_podversion` **Returns** the version of the iPod the application is running on, one of the `TTK_POD_*` macros defined above. The value is cached after it is checked the first time, so this function is very fast.

`ttk_get_screensize` inserts the screen width into `*w`, the screen height into `*h`, and the bit depth into `*bpp`, assuming none of those pointers are `NULL`. (Any null pointers are skipped over.)

`ttk_set_emulation` sets the screen to be $w \times h \times bpp$ pixels if the application is running on X11. It is ignored on the iPod. *This must be called* ***before*** `ttk_init()`.

`ttk_click` produces the "click" sound that occurs on the iPod when you scroll through menus etc. If running under X11, this function has no effect.

`ttk_run` starts up event processing for a TTK application. Events will continue to be processed and sent to widgets, even if said widgets open new windows and cause new widgets to come into focus. The event processing will end when an event handler returns `TTK_EV_DONE`, and the value set with `TTK_EV_RET` will be returned.

You are free to call `ttk_run()` from within an event handler; the canonical example of this would be a menu option that opens a dialog box and then does something with the dialog's return value. If the dialog were to be opened just before the event handler returned, the postprocessing on whatever the user selected would be much more cumbersome. A better way would be to show the dialog window, have one of its event handlers returning `TTK_EV_DONE`, and then run a sub-`ttk_run()` to "do the dialog." The return value could then be read easily, and appropriate action taken. (This is the technique used by `podzilla`.)

4.2 Window Management

```
TWindow *ttk_new_window();
void ttk_free_window (TWindow *win);
void ttk_show_window (TWindow *win);
void ttk_draw_window (TWindow *win);
```

```
void ttk_set_popup (TWindow *win);
void ttk_move_window (TWindow *win, int offset, int whence);
int ttk_hide_window (TWindow *win);
void ttk_popup_window (TWindow *win);
void ttk_window_title (TWindow *win, const char *str);
void ttk_window_show_header (TWindow *win);
void ttk_window_hide_header (TWindow *win);

#define ttk_popdown_window(w) ttk_hide_window(w)
#define ttk_window_set_title(w,s) ttk_window_title(w,s)
```

These functions are used to do practically everything with windows.

`ttk_new_window` creates a new window and returns it. The window title is set to be "TTK," and its geometry is set to cover the entire screen. Other values are set to a sensible default.

`ttk_free_window` deallocates the memory used by a window. If the window is shown onscreen, it is hidden first. **All widgets in the window are also freed.**

`ttk_show_window` shows a window onscreen.

`ttk_hide_window` hides it. **Returns** the number of windows hidden (can be more than 1 in degenerate cases or 0 if `win` is not shown). `win` will be hidden even if it is the last window onscreen; however, in that case `ttk_run()` will exit very soon.

`ttk_draw_window` draws a window to the screen in exactly the manner used by `ttk_run()`. It is useful mainly for multi-window interfaces, where lower windows need to be drawn though they are not receiving input, and in combination with `ttk_gfx_update()` for animations in the course of one event handler.

`ttk_set_popup` sets up a window to become a popup window. All widgets should already be in the window. Widgets are moved and window size is adjusted so that all widgets are moved as close as possible (relative to one another) to the upper-left hand corner of the window, the window's width and height are made as small as possible, and x and y coordinates are chosen so the window will be in the middle of the screen.

`ttk_popup_window` calls `ttk_set_popup()` followed by `ttk_show_window()`.

`ttk_move_window` moves `win` to position `offset` in the window stack (with the top window 0) if `whence` is `TTK_MOVE_ABS`; or moves `win` down `offset` windows in the window stack if `whence` is `TTK_MOVE_REL`; or moves `win offset` windows *away* from the end of the window stack if `whence` is `TTK_MOVE_END`.

`ttk_window_title` (or `ttk_window_set_title`) sets the title of `win` to a copy of `str`, sets a flag in `win` so the title will be freed when the window is, and causes the header to be redrawn. If you don't need all of this, you can just do `win->title = str`.

`ttk_window_show_header` **and** `ttk_window_hide_header` cause the header to be either shown or not shown, respectively, when this window is at the top of the stack. This is useful for screensavers, and for widgets that need as much screen space as possible.

```
enum ttk_justification {
        TTK_TEXT_CENTER = 0,
        TTK_TEXT_LEFT,
        TTK_TEXT_RIGHT
};
void ttk_header_set_text_position( int x );
void ttk_header_set_text_justification( enum ttk_justification j );
```

These two functions set the behavior of the text rendering for the header bar.

`ttk_header_set_text_justification` sets how the text is positioned with relation of the position. Use the values defined in the enum as parameters for this. Read the next item for more description.

`ttk_header_set_text_position` sets where the text gets drawn from. If Centered, this is the center point of the text. If Left justified, this is the leftmost point of the text. If Right justified, this is the rightmost point of the text.

4.3 Widget Management

```
TWidget *ttk_new_widget (int x, int y);
void ttk_free_widget (TWidget *wid);
TWindow *ttk_add_widget (TWindow *win, TWidget *wid); // returns win
int ttk_remove_widget (TWindow *win, TWidget *wid);
void ttk_widget_set_fps (TWidget *wid, int fps);
void ttk_widget_set_inv_fps (TWidget *wid, int fps_m1);
void ttk_widget_set_timer (TWidget *wid, int ms);

void ttk_add_header_widget (TWidget *wid);
void ttk_remove_header_widget (TWidget *wid);
```

These functions allow you to manage the widgets in your applications.

`ttk_new_widget` creates a new widget positioned at (x, y) and returns it. The event handlers for this widget are set to do-nothing but valid functions that return `TTK_EV_UNUSED`. *You should* ***never*** *set a widget's event handler to* `NULL`.

`ttk_free_widget` removes `wid` from its window if it has one, calls `wid->destroy()`, and frees all memory associated with it.

`ttk_add_widget` adds `wid` to the widget list of `win`, setting the widget's `win` pointer in the process.

`ttk_remove_widget` removes `wid` from `win`'s widget list, if it is in it, and sets `wid`'s `win` pointer to `NULL`.

`ttk_widget_set_fps` sets up `wid->frame()` to be called `fps` times per second while events are being processed.

`ttk_widget_set_inv_fps` sets up `wid->frame()` to be called *every* `fps_m1` *seconds*. The reason it's called `inv_fps` is that it works somewhat like a theoretical `ttk_widget_set_fps(wid, 1/fps_m1)` should work, except that it obviously doesn't use floating-point.

`ttk_widget_set_timer` sets up `wid->timer()` to be called every `ms` milliseconds while events are being processed.

`ttk_add_header_widget` adds `wid` to the list of widgets to be drawn in the header. The only events processed for these widgets will be frame and timer, and they will be drawn about once every time you open a new window, unless you use one of the periodic event handlers to set the dirty flag more regularly.

`ttk_remove_header_widget` removes `wid` from the list of header widgets.

4.4 Events and Input

```
void ttk_set_global_event_handler (int (*fn)(int ev, int earg, int time));
void ttk_set_global_unused_handler (int (*fn)(int ev, int earg, int time));
int ttk_button_pressed (int button);

int ttk_input_start_for (TWindow *win, TWidget *method);
void ttk_input_move_for (TWindow *win, int x, int y);
void ttk_input_size_for (TWindow *win, int *w, int *h);

int ttk_input_start (TWidget *method);
void ttk_input_move (int x, int y);
void ttk_input_size (int *w, int *h);
void ttk_input_char (char ch);
void ttk_input_end();
```

The first two functions let you control event handling at a global level.

Each handler function has three parameters. The first is the event type, one of `TTK_BUTTON_DOWN`, `TTK_BUTTON_UP`, or `TTK_SCROLL`. The second is the button that was pushed, or the distance that was scrolled. Finally, if the event was a button-up event, the third argument indicates the amount of time for which the button was pressed.

The global event handler is called before widget-specific handlers, and acts as a gateway to them: it can do whatever it wants, and if it returns 1 the event will not be pursued further.

The global unused handler is called *after* a widget-specific handler that returned `TTK_EV_UNUSED`; its return code is like that of a widget event handler, consisting of a combination of `TTK_EV_*` flags.

The other one—`ttk_button_pressed`—returns the time in milliseconds since the program started that the specified button was pressed, or 0 if it is not currently being held down.

Finally, we have the text input functions. They are as follows.

`ttk_input_start` sets up `method` to be a text-input method, as described in section 1.5(?). The `w` and `h` members of the widget should be set, but `x` and `y` should not be. These will be set to place the widget in the bottom of the screen, to the right if it is not the full width of the screen. **Returns** the height of the text input method.

`ttk_input_move` repositions the text input method currently in use to be at (x, y). If text input is not started, this does nothing.

`ttk_input_size` places the width of the current input widget in `*w`, and the height in `*h`, if text input is started. If either pointer is NULL, it will be ignored.

`ttk_input_*_for` are variants of the past three functions that allow you to specify what window to work with, instead of assuming the top window currently shown. This lets you e.g. set up a window for TI before showing it.

`ttk_input_char` queues `ch` to be sent to the focused widget's `input` event handler. This may be called multiple-times for multiletter sequences.

`ttk_input_end` stops text input and **frees the text input method**.

4.5 iPod-Specific Functions

```
void ttk_update_lcd (int xstart, int ystart, int xend, int yend,
                     unsigned char *data);
```

These functions let you do low-level hardware things with the iPod. Some of them do not work very well on X11.

`ttk_update_lcd` dumps a chunk of raw data to the iPod's LCD; the data should already be in the appropriate format for the iPod running. The `data` block should be the size of the whole screen, but only the region from $(xstart, ystart)$ to $(xend, yend)$ will be updated. This function does not work on X11.

4.6 Appearance Functions

These functions manage the appearance subsystem. See Section 6.3 for details about how the properties are used in the drawing functions.

```
void ttk_ap_load (const char *filename);
TApItem *ttk_ap_get (const char *prop);
TApItem *ttk_ap_getx (const char *prop);
void ttk_ap_hline (ttk_surface srf, TApItem *ap, int x1,
                   int x2, int y);
void ttk_ap_vline (ttk_surface srf, TApItem *ap, int x,
                   int y1, int y2);
void ttk_ap_rect (ttk_surface srf, TApItem *ap, int x1,
                  int y1, int x2, int y2);
void ttk_ap_fillrect (ttk_surface srf, TApItem *ap, int x1,
                      int y1, int x2, int y2);
```

`ttk_ap_load` loads a color scheme from `file`, which must be a full path (not relative to the schemes directory), over the current one. You should probably do `ttk_epoch++` directly after this, if you want the changes to take effect immediately.

`ttk_ap_get` searches for the first instance of the appearance property with key `prop`. If it is found, a pointer to an internal structure for it is returned. If not, NULL is returned. You should not modify the returned structure.

`ttk_ap_getx` is a version of `ttk_ap_get` for use in situations when you are immediately dereferencing the result; if the specified `prop` is not found, a dummy structure is returned instead of NULL.

`ttk_ap_hline` draws a horizontal line from (x_1, y) to (x_2, y) using the specified appearance item `ap`, on `srf`.

`ttk_ap_vline` draws a vertical line from (x, y_1) to (x, y_2) using the specified appearance item `ap`, on `srf`.

`ttk_ap_rect` **and** `ttk_ap_fillrect` draw unfilled and filled rectangles, respectively, from (x_1, y_1) to (x_2, y_2) using appearance item `ap` on surface `srf`.

4.7 Graphics Functions

There are a lot of graphics-related functions, so this section is subdivided.

4.7.1 Surface and Color Functions

```
ttk_surface ttk_new_surface (int w, int h, int bpp);
ttk_surface ttk_load_image (const char *path);
void ttk_free_surface (ttk_surface srf);
void ttk_surface_get_dimen (ttk_surface srf, int *w, int *h);
ttk_surface ttk_scale_surface (ttk_surface srf, float factor);
void ttk_blit_image (ttk_surface src, ttk_surface dst, int dx, int dy);
void ttk_blit_image_ex (ttk_surface src, int sx, int sy, int sw, int sh,
                        ttk_surface dst, int dx, int dy);

ttk_color ttk_makecol (int r, int g, int b);
ttk_color ttk_makecol_ex (int r, int g, int b, ttk_surface srf);
void ttk_unmakecol (ttk_color col, int *r, int *g, int *b);
void ttk_unmakecol_ex (ttk_color col, int *r, int *g, int *b, ttk_surface srf);
ttk_color ttk_mapcol (ttk_color col, ttk_surface src, ttk_surface dst);
```

These functions let you create, destroy, and manipulate surfaces and colors. (The terms "image" and "surface" may be used interchangeably in TTK.)

`ttk_new_surface` creates a new surface of the specified width, height, and bit depth, and fills it with white.

`ttk_load_image` creates a new surface whose contents and dimensions are those of the specified image file. PNG is definitely supported; other formats may or may not be.

`ttk_free_surface` frees memory associated with a surface, whether it was created by `ttk_new_surface` or `ttk_load_image`.

`ttk_surface_get_dimen` places the width and height of `srf` into `*w` and `*h` respectively.

`ttk_scale_surface` scales `srf` by `factor` to create a new surface. **You must free the returned surface separately from `srf`.**

`ttk_blit_image` copies the full contents of `src` (or as much as will fit) in a rectangle on `dst` whose upper-left hand corner is at (dx, dy).

`ttk_blit_image_ex` copies the rectangle $(sx, sy)sw \times sh$ on `src` to (dx, dy) on `dst`.

`ttk_makecol` creates a `ttk_color` value suitable for drawing on the screen, window surfaces, or anything with the same bit depth.

`ttk_makecol_ex` creates a `ttk_color` value suitable for drawing on `srf`.

`ttk_unmakecol` returns the RGB values used by a call to `ttk_makecol`.

`ttk_unmakecol_ex` returns the RGB values used by a call to `ttk_makecol_ex` on `srf`.

`ttk_mapcol` maps a color value intended for `src` into one intended for `dst`.

4.7.2 Graphics Context Functions

```
ttk_gc ttk_new_gc();
ttk_gc ttk_copy_gc (ttk_gc other);
ttk_color ttk_gc_get_foreground (ttk_gc gc);
ttk_color ttk_gc_get_background (ttk_gc gc);
ttk_font ttk_gc_get_font (ttk_gc gc);
void ttk_gc_set_foreground (ttk_gc gc, ttk_color fgcol);
void ttk_gc_set_background (ttk_gc gc, ttk_color bgcol);
```

```
void ttk_gc_set_font (ttk_gc gc, ttk_font font);
void ttk_gc_set_usebg (ttk_gc gc, int flag);
void ttk_gc_set_xormode (ttk_gc gc, int flag);
void ttk_free_gc (ttk_gc gc);
```

These functions are used to create, destroy, and modify the properties of graphics contexts.

`ttk_new_gc` creates a new graphics context, with all values set to 0.

`ttk_copy_gc` creates a copy of `other` that must be freed separately.

`ttk_free_gc` deallocates all memory associated with `gc`.

`ttk_gc_get_foreground` and friends retrieve the relevant property of the GC.

`ttk_gc_set_foreground` sets the foreground color for drawing operations with `gc`.

`ttk_gc_set_background` sets `gc`'s background color, which is only used when drawing text, and only when the usebg flag is set.

`ttk_gc_set_font` sets the font to use when drawing text with `gc`.

`ttk_gc_set_usebg` sets the usebg flag for `gc`, which determines whether or not a background is drawn when drawing text.

`ttk_gc_set_xormode` sets the xormode flag for `gc`, which determines whether or not drawing operations invert pixels instead of setting them. This flag is only guaranteed to work for the graphics primitive `ttk_fillrect_gc`.

4.7.3 Graphics Primitives

```
void ttk_pixel (ttk_surface srf, int x, int y, ttk_color col);
void ttk_pixel_gc (ttk_surface srf, ttk_gc gc, int x, int y);

void ttk_line (ttk_surface srf, int x1, int y1, int x2, int y2, ttk_color col);
void ttk_line_gc (ttk_surface srf, ttk_gc gc, int x1, int y1, int x2, int y2);

void ttk_rect (ttk_surface srf, int x1, int y1, int x2, int y2, ttk_color col);
```

```
void ttk_rect_gc (ttk_surface srf, ttk_gc gc, int x, int y, int w, int h);
void ttk_fillrect (ttk_surface srf, int x1, int y1, int x2, int y2, ttk_color col);
void ttk_fillrect_gc (ttk_surface srf, ttk_gc gc, int x, int y, int w, int h);

void ttk_poly (ttk_surface srf, int nv, short *vx, short *vy, ttk_color col);
void ttk_poly_gc (ttk_surface srf, ttk_gc gc, int n, ttk_point *v);
void ttk_fillpoly (ttk_surface srf, int nv, short *vx, short *vy, ttk_color col);
void ttk_fillpoly_gc (ttk_surface srf, ttk_gc gc, int n, ttk_point *v);

void ttk_ellipse (ttk_surface srf, int x, int y, int rx, int ry, ttk_color col);
void ttk_ellipse_gc (ttk_surface srf, ttk_gc gc, int x, int y, int rx, int ry);
void ttk_fillellipse (ttk_surface srf, int x, int y, int rx, int ry, ttk_color col);
void ttk_fillellipse_gc (ttk_surface srf, ttk_gc gc, int x, int y, int rx, int ry);

void ttk_bitmap (ttk_surface srf, int x, int y, int w, int h, unsigned short *bits, ttk_color co
void ttk_bitmap_gc (ttk_surface srf, ttk_gc gc, int x, int y, int w, int h, unsigned short *bits
```

These functions' functions are, for the most part, obvious from their names; only the non-obvious will be documented here.

`ttk_*rect` use the coordinates of upper-left and one past the bottom-right corners; `ttk_*rect_gc` use the upper-left corner and width and height. **The point (x_2, y_2) and the lines containing it are not drawn.**

`ttk_*poly` use separate arrays for X and Y coordinates; `ttk_*poly_gc` use one array of `ttk_point` structures. The poly (polyline) functions do not connect the first point to the final point; that is to say that they can create open polyline graphics as well as closed polygon graphics.

`ttk_bitmap` draws an array of bits on the surface at the specified (x, y). The bitmap format is an array of `unsigned shorts`, filled **from MSB to LSB**; that is, the MSB of the first short will be at (x, y). You should have a whole number of shorts per row; unused bits in the least-significant part of one at the end of a row should be set to 0. Bits set indicate pixels colored `col`; bits cleared will not be drawn in any color. *You must specify a correct width and height.* If you do not, your image will look ... odd, very odd.

As an example, here's a 4×4 "ball":

```
static unsigned short pong_ball[] = {
    0x6000, // . X X .
    0xf000, // X X X X
    0xf000, // X X X X
```

```
    0x6000, // . X X .
    0       // not necessary
};
```

4.7.4 Gradient Functions

```
void ttk_hgradient( ttk_surface srf, int x1, int y1, int x2, int y2,
                        ttk_color left, ttk_color right );
void ttk_vgradient(ttk_surface srf, int x1, int y1, int x2, int y2,
                        ttk_color top, ttk_color bottom );
```

These two behave like `ttk_fillrect` above. The only addition is `left`, `right`, `top` and `bottom`, which define the colors used in the gradient. hgradient draws a gradient of vertical lines that progress horizontally, vgradient progresses vertically.

For example, `ttk_hgradient` draws a rectangle, similarly to `ttk_fillrect` but the contents are filled with a gradient that is "left" color on the left side of the rectangle, and "right" color on the right side of the rectangle.

```
typedef struct _gradient_node {
        ttk_color start;
        ttk_color end;
        ttk_color gradient[256];
        struct _gradient_node * next;
} gradient_node;
void ttk_gradient_clear( void );
gradient_node * ttk_gradient_find( ttk_color start, ttk_color end );
gradient_node * ttk_gradient_find_or_add( ttk_color start, ttk_color end );
```

These functions deal with the creation and retrieval of gradients, which are stored in a simple linked list. These functions are internally used by `ttk_hgradient` and `ttk_vgradient`, as described above.

`ttk_gradient_find` will search the internal list for the specified color gradient. If it is found, it will return the `gradient_node` that matches. If it is not found, it will return NULL. `ttk_gradient_find_or_add` will search the internal list for the specified color gradient. If it is found, it will return

the `gradient_node` that matches. If it is not found, it will create a new node, populate it with the requested gradient array, prepend it onto the main list, and return the new node.

The gradient itself is stored as 256 elements that progress from "start" to "end", color-wise. Those are in the "gradient" array in the struct.

4.7.5 Text and Font Functions

```
ttk_fontinfo *ttk_get_fontinfo (const char *name, int size);
ttk_font ttk_get_font (const char *name, int size);
void ttk_done_fontinfo (ttk_fontinfo *fi);
void ttk_done_kfont (ttk_font f);

void ttk_text (ttk_surface srf, ttk_font fnt, int x, int y,
               ttk_color col, const char *str);
void ttk_text_lat1 (ttk_surface srf, ttk_font fnt, int x, int y,
                    ttk_color col, const char *str);
void ttk_text_uc16 (ttk_surface srf, ttk_font fnt, int x, int y,
                    ttk_color col, const uc16 *str);
void ttk_text_gc (ttk_surface srf, ttk_gc gc, int x, int y,
                  const char *str);

int ttk_text_width (ttk_font fnt, const char *str);
int ttk_text_width_lat1 (ttk_font fnt, const char *str);
int ttk_text_width_uc16 (ttk_font fnt, const uc16 *str);
int ttk_text_width_gc (ttk_gc gc, const char *str);
int ttk_text_height (ttk_font fnt);
int ttk_text_height_gc (ttk_gc gc);
```

These functions are used to get fonts and draw text on surfaces.

`ttk_get_fontinfo` retrieves a `ttk_fontinfo` structure for the closest match that can be found to `name` at `size`.

`ttk_get_font` returns `ttk_get_fontinfo(name,size)->f`; that is, it returns the actual font, not the font information structure. Unless you need the extra information, you should probably use this function.

`ttk_done_fontinfo` unloads the font described by `fi`; it will be reloaded the next time it is requested. This is rather a good idea, since fonts can take upwards of 600k of RAM. Reference-counting is employed; thus, you must call `ttk_get_fontinfo()` and `ttk_done_fontinfo()` an equal number of times.

`ttk_done_font` unloads the font `f`; see the documentation for `ttk_done_fontinfo()`.

`ttk_text_*` draw the string `str` colored `col` using font `fnt` at (x, y) on `srf`. The (x, y) coordinates refer to the **top-left corner** of the text.

`ttk_text_gc` draws the string `str` with its upper-left hand corner at (x, y) on `srf`, using information in `srf`.

`ttk_text_width_*` return the width of `str` in the font `fnt`; `ttk_text_width_gc` does the same thing for the font selected in the graphics context `gc`.

`ttk_text_height` returns the height of one line of text in `fnt`; `ttk_text_height_gc` is the analagous graphics-context version. (All characters in TTK fonts are the same height.)

Character encoding is UTF-8 by default. To draw Latin-1 text, call `ttk_text_lat1`; to draw Unicode text, call `ttk_text_uc16`. The `uc16` type is equivalent to an `unsigned short`.

4.8 Miscellaneous

This section contains all the functions that didn't fit anywhere else.

```
void ttk_gfx_update (ttk_surface srf);
int ttk_getticks();
void ttk_delay (int ms);

ttk_timer ttk_create_timer (int ms, void (*callback)());
void ttk_destroy_timer (ttk_timer tim);

void ttk_set_scroll_multiplier (int num, int denom);
void ttk_set_transition_frames (int frames);
```

```
void ttk_set_clicker (void (*clickfn)());
  void ttk_click();
```

`ttk_gfx_update(ttk_screen->srf)` will cause the current screen image to be actually drawn on the LCD. This is necessary on SDL, among other things, but you'll never have to use it yourself unless you want some special effects.

`ttk_getticks` returns the number of milliseconds that have passed since the program started. `ttk_delay` sleeps for `ms` milliseconds.

`ttk_create_timer` creates and returns a new timer set to call callback once, `ms` milliseconds from now. The returned `ttk_timer` can be used as an argument to `ttk_destroy_timer` to stop it from firing.

`ttk_set_scroll_multiplier` is the "wheel sensitivity" setting; it sets things up so a physical scroll of `denom` units will be treated as one of `num` units by all widgets. (Try to keep the denominator low; something like 127/128 would fire 127 scroll events at once, every time you scrolled continuously for 128!)

`ttk_set_transition_frames` sets the number of frames used by the window transition; 1 or less turns the transition off. `ttk_set_clicker` sets the function used when event handlers return `TTK_EV_CLICK`; its default is `ttk_click()`, which clicks the piezo on the iPod and does nothing on the desktop.

4.9 Nano-X Emulation

This section contains the complete list of all emulated Nano-X functions. They are not documented here; use a Nano-X API reference for that. Functions not in this list are unimplemented. Arguments named `_unused`*n* are ignored.

```
int t_GrOpen();
void t_GrClose();
#define GrFlush() // not needed
#define GrSelectEvents(w,e) // not needed
int t_GrPeekEvent (t_GR_EVENT *ev);
#define GrCheckNextEvent(e) t_GrGetNextEventTimeout(e,0)
```

```
int t_GrGetNextEventTimeout (t_GR_EVENT *ev, int timeout);
void t_GrGetScreenInfo (t_GR_SCREEN_INFO *inf);

t_GR_WINDOW_ID t_GrNewWindow (int _unused1, int x, int y, int w, int h,
                              int _unused2, int _unused3, int _unused4);
t_GR_WINDOW_ID t_GrNewWindowEx (int _unused1, const char *title, int _unused3,
                                int x, int y, int w, int h, int _unused4);
void t_GrDestroyWindow (t_GR_WINDOW_ID w);
void t_GrMapWindow (t_GR_WINDOW_ID w);
void t_GrUnmapWindow (t_GR_WINDOW_ID w);
void t_GrResizeWindow (t_GR_WINDOW_ID win, int w, int h);
void t_GrMoveWindow (t_GR_WINDOW_ID win, int x, int y);
#define GrSetWindowBackgroundColor(w,c) // not implemented
void t_GrClearWindow (t_GR_WINDOW_ID w, int _unused);
void t_GrGetWindowInfo (t_GR_WINDOW_ID w, t_GR_WINDOW_INFO *inf);
t_GR_WINDOW_ID t_GrGetFocus();
t_GR_WINDOW_ID t_GrNewPixmap (int w, int h, int _unused1);

t_GR_GC_ID t_GrNewGC();
t_GR_GC_ID t_GrCopyGC (t_GR_GC_ID other);
void t_GrDestroyGC (t_GR_GC_ID gc);
void t_GrSetGCForeground (t_GR_GC_ID gc, t_GR_COLOR col);
void t_GrSetGCBackground (t_GR_GC_ID gc, t_GR_COLOR col);
void t_GrSetGCUseBackground (t_GR_GC_ID gc, int flag);
void t_GrSetGCMode (t_GR_GC_ID gc, int mode);
void t_GrSetGCFont (t_GR_GC_ID gc, t_GR_FONT_ID font);
void t_GrGetGCInfo (t_GR_GC_ID gc, t_GR_GC_INFO *inf);

void t_GrPoint (t_GR_DRAW_ID dst, t_GR_GC_ID gc, int x, int y);
void t_GrLine (t_GR_DRAW_ID dst, t_GR_GC_ID gc, int x1, int y1, int x2, int y2);
void t_GrRect (t_GR_DRAW_ID dst, t_GR_GC_ID gc, int x, int y, int w, int h);
void t_GrFillRect (t_GR_DRAW_ID dst, t_GR_GC_ID gc, int x, int y, int w, int h);
void t_GrPoly (t_GR_DRAW_ID dst, t_GR_GC_ID gc, int n, t_GR_POINT *v);
void t_GrFillPoly (t_GR_DRAW_ID dst, t_GR_GC_ID gc, int n, t_GR_POINT *v);
void t_GrEllipse (t_GR_DRAW_ID dst, t_GR_GC_ID gc, int x, int y, int rx, int ry);
void t_GrFillEllipse (t_GR_DRAW_ID dst, t_GR_GC_ID gc, int x, int y, int rx, int ry);
#define GrArc(x,y,a,b,c,d,e,f,g,h,m) // not implemented
void t_GrCopyArea (t_GR_DRAW_ID dst, t_GR_GC_ID gc, int dx, int dy, int sw, int sh,
                   t_GR_DRAW_ID src, int sx, int sy, int _unused1);
void t_GrBitmap (t_GR_DRAW_ID dst, t_GR_GC_ID gc, int x, int y, int w, int h,
                 t_GR_BITMAP *imgbits);
t_GR_IMAGE_ID t_GrLoadImageFromFile (const char *file, int _unused1);
void t_GrDrawImageToFit (t_GR_DRAW_ID dst, t_GR_GC_ID gc, int x, int y, int w, int h,
                         t_GR_IMAGE_ID img);
void t_GrFreeImage (t_GR_IMAGE_ID img);
```

```
void t_GrText (t_GR_DRAW_ID dst, t_GR_GC_ID gc, int x, int y, const void *str,
               int count, int _unused);

t_GR_FONT_ID t_GrCreateFont (const char *path, int size, struct t_GR_LOGFONT *_unused1);
void t_GrDestroyFont (t_GR_FONT_ID font);
void t_GrGetGCTextSize (t_GR_GC_ID gc, const void *str, int count, int flags,
                        int *width, int *height, int *base);

t_GR_TIMER_ID t_GrCreateTimer (t_GR_WINDOW_ID win, int ms);
void t_GrDestroyTimer (t_GR_TIMER_ID tim);
```

Chapter 5

Widgets

TTK uses widgets to implement its user interface; to spare everyone the trouble of inventing a wheel (and ensure that no one decides to invent a *square* wheel), four builtin widgets are provided. Each is considerably complex and has a lot of features. They are described here.

5.1 Menu

The menu widget provides a flexible framework for a hierarchial menu system like that used by the standard Apple firmware on the iPod. A menu is basically a collection of items; various internal structures are kept, but the only one you need to concern yourself with is the item structure. It will be described below. There are two types of items: "selecting" items and "setting" items. Selecting items are those that launch a new window; setting items are those, like the ones in the Apple firmware's settings menu, that change a setting when selected. You will generally use different combinations of members in the menu structure for each type of item.

5.1.1 The Widget

```
typedef struct ttk_menu_item
{
    const char *name;
```

```
    struct {
        TWindow *(*makesub) (struct ttk_menu_item *item);
        TWindow *sub;
    };
    int flags;
    void *data;
    const char **choices;
    int cdata;
    void (*choicechanged)(struct ttk_menu_item *item, int cdata);
    int (*choiceget)(struct ttk_menu_item *item, int cdata);
    int choice;
    int (*visible)(struct ttk_menu_item *item);
    int free_name;
    int free_data;

    /* for groups */
    char * group_name;
    int group_flags;

    /* readonly */ int menuwidth, menuheight;
} ttk_menu_item;
```

Before we discuss the specific members, a word about the structure of this structure is appropriate. The order of the members and such is designed to allow easy initialization in a static array. For instance:

```
  { "My Nice Menu Item", {my_menu_handler}, TTK_MENU_ICON_SUB, my_data }
```

As such, the members are not in much order. If you want to make a menu item a settings item instead of a selection item (to be discussed later), the thought is that you will indicate the appropriate members with GNU `.memb = val` syntax, so as to avoid having to specify 0 for all the inapplicable members before it.

Ease of definition also prompted the seemingly crazy anonymous-structure idea, since in the majority of cases you want to define *either* a window *or* a handler to produce one; however, a union is inappropriate, since the menu widget uses both fields at once internally. As such, an anonymous structure

is used; while this requires you to do `{my_menu_handler}` for a handler function or `{.sub=my_window}` for a (less common) already-created window, it is much less confusing than having an extra zero in the list for the unused choice.

And now, for the members themselves...

`name` is the name of the menu item: what will be displayed on the screen. If the name is longer than the screen width, it will be abbreviated when unselected and scrolled horizontally when selected.

⟨*anon*⟩.makesub is a pointer to a handler function that will be called when an item is first selected, and will return a window that will be displayed from then on when that item is selected.

⟨*anon*⟩`.sub` is a pointer to a window that will be displayed when this item is selected. Usually, you want to leave this as 0, and use a handler function.

`flags` is a bitwise OR of the flags enumerated below.

`data` is data for the handler function, or whatever else you want to use it for.

`choices` is a NULL-terminated array of strings that list the choices for this setting. Its non-NULL presence implies that this menu item is a setting item.

`cdata` is some more data, this time an integer that you can use for whatever you want

choicechanged is a function that is called whenever the user changes the setting for this item. It may be useful if the setting concerns something you want the user to have immediate feedback on.

choiceget is a function called once, when the item is added to the menu or the menu is initialized, that is expected to return the current setting for this item. If this is NULL, the initial value of `choice` will be used.

`choice` stores the current choice for the setting of this item, as an index into `choices[]`.

visible is a function that will be called quite often, if defined, with the item structure as argument. It should return nonzero if the item is to be displayed. If it is set to NULL, the item will always be displayed. Changes in displayedness may take about a second to show onscreen.

`menuwidth` **and** `menuheight` contain the width and the height of the menu this item is in; this is used by the submenu handler function, among others.

`free_name` **and** `free_data` indicate whether the name and the data fields should be respectively freed when the menu is.

`group_name` **and** `group_flags` determine what group the menu item should be associated with in the menu. When sorted, the groups are first alphabetized, then the items within them will be sorted alphabetically. If there is none specified, it will be in a group which will be sorted after all other groups. The group flags can be used to specify whether or not the group name/title text is visible. Other options may be available in the future.

5.1.2 Flags

The flags are:

```
#define TTK_MENU_ICON_SUB 01
#define TTK_MENU_ICON_EXE 02
#define TTK_MENU_ICON_SND 04
#define TTK_MENU_ICON     07
#define TTK_MENU_MADESUB  010
/* and others you don't need to know about */
```

The `TTK_MENU_ICON_*` flags indicate which icon, if any, you want displayed on the right of your menu item. The icon will flash when the item is selected. You can use `TTK_MENU_ICON` as a bitmask to check for the presence of *any* icon; you shouldn't set it yourself.

If you specified a sub and not a makesub, you need to set the `TTK_MENU_MADESUB` flag. Basically, when a menu item is selected, this flag is checked: if it is set, the sub window is opened; if it is not, makesub is called.

5.1.3 Window Lifetimes

This is its own subsection because it's *important*.

The menu handler will generally only call makesub *once*, and use the window it returns from then on. If you wish to avoid this behavior, set `win->data = 0x12345678` at some point; the next time that item is selected, the window will be freed and recreated from makesub.

If, however, the "window" is set to a directive like `TTK_MENU_DONOTHING` (as explained below), the returned value will *not* be cached; the menu handler function will be called again next time the item is selected.

5.1.4 The Menu Handler

The menu handler—that ⟨*anon*⟩.makesub function—can do one of the following things. Those that are not just creating a new window and returning it involve `TTK_MENU_*` constants, basically integers cast to `TWindow *`. These will be called "directives" and are saved just like normal window returns.

Make and open new window. The handler should create the window and return it, but should not show it onscreen.

No change in open window. The handler should do whatever it wants and return `TTK_MENU_DONOTHING`.

"Back" item. The handler should do whatever it needs to and return `TTK_MENU_UPONE`; the menu's window will be closed.

"Main Menu" item. The handler should return `TTK_MENU_UPALL`; this will close all windows from here up that are pure menu windows (without change in the draw handler) and are set closeable.

Window already opened by handler's actions. This is the case, for instance, with legacy functions that both create and open a window. The handler should set `item->sub = ttk_windows->w` and return `TTK_MENU_ALREADYDONE`.

Menu event handler should return `TTK_EV_DONE` to exit. Return `TTK_MENU_QUIT`. You have no control over the return value from `ttk_run()`.

Menu window should be hidden, child window shown in its place. Show the child window with `ttk_show_window()` and return `TTK_MENU_REPLACE` from the menu handler.

+ Window should be reopened every time the item is selected. The handler should, in addition, set the window's `data` member to `0x12345678`.

5.1.5 Menu Data Assemblers

Since the menu handler has only the `data` pointer to work with, and many widgets require more than one parameter, and probably not a `void *`, menuable widgets in TTK provide two functions. The first is the menu handler, and is generally called `ttk_mh_`*name*, where *name* is something descriptive like `textarea`. The second is something called a **menu data assembler**: it is named `ttk_md_`*name*. Taking the textarea example, which needs two parameters (the text and the number of pixels between lines), a menu item would look like this:

```
{ "My Nice Textarea", {ttk_mh_textarea}, 0, ttk_md_textarea (the_text, 14) }
```

This dual-function `mh`/`md` usage yields both simplicity of definition and simplicity of programming, and its use should be encouraged.

5.1.6 Functions

```
TWidget *ttk_new_menu_widget (ttk_menu_item *items, ttk_font font, int w, int h);
ttk_menu_item *ttk_menu_get_item (TWidget *_this, int i);
ttk_menu_item *ttk_menu_get_item_called (TWidget *_this, const char *s);
ttk_menu_item *ttk_menu_get_selected_item (TWidget *_this);
void ttk_menu_set_closeable (TWidget *_this, int closeable);
void ttk_menu_sort (TWidget *_this);
void ttk_menu_sort_my_way (TWidget *_this, int (*cmp)(const void *, const void *));
void ttk_menu_append (TWidget *_this, ttk_menu_item *item);
void ttk_menu_insert (TWidget *_this, ttk_menu_item *item, int pos);
void ttk_menu_remove (TWidget *_this, int pos);
void ttk_menu_remove_by_ptr (TWidget *_this, ttk_menu_item *item);
void ttk_menu_remove_by_name (TWidget *_this, const char *name);
void ttk_menu_item_updated (TWidget *, ttk_menu_item *item);
void ttk_menu_updated (TWidget *_this);
```

```
TWindow *ttk_mh_sub (struct ttk_menu_item *item);
void *ttk_md_sub (struct ttk_menu_item *submenu);

/*
 * The following are standard handler functions. You should call them only from your own
 * overridden handlers. They will not be documented.
 */
void ttk_menu_draw (TWidget *_this, ttk_surface srf);
int ttk_menu_scroll (TWidget *_this, int dir);
int ttk_menu_down (TWidget *_this, int button);
int ttk_menu_frame (TWidget *_this);
void ttk_menu_free (TWidget *_this);
```

`ttk_new_menu_widget` creates and returns a new menu widget created from the list of items in `items`. The menu will be displayed with the font `font`, but submenus will be displayed with `ttk_menufont` for various reasons. The menu will be `w` × `h` pixels in dimension, located at $(0, 0)$. **If `items` = 0, all items later added to the menu will be freed when it is freed. If not, they won't.**

`ttk_menu_get_item` returns item number `i` in the supplied menu.

`ttk_menu_get_item_called` returns the first item whose name compares equal to `s`.

`ttk_menu_get_selected_item` returns the selected item in the supplied menu.

`ttk_menu_set_closeable` sets the closeable flag for the specified menu. When this flag is cleared, the menu may not be closed by pressing Menu. When it is set (the default), it may.

`ttk_menu_sort` sorts the items in the supplied menu by case-sensitive name.

`ttk_menu_sort_my_way` sorts the items in the supplied menu using the comparator function `cmp`, of the sort supplied to `qsort`. The parameters are actually `ttk_menu_item **`s, and should be cast as such.

`ttk_menu_append` adds `item` to the end of the specified menu. Variable allocation for internal structures is handled automatically.

`ttk_menu_insert` inserts `item` into position `pos` in the menu. If `pos` is greater than the number of items in the menu, this function behaves like `ttk_menu_append`.

`ttk_menu_remove` removes item number `pos` in the menu.

`ttk_menu_remove_by_ptr` removes the first instance, if any, of `item` in the menu.

`ttk_menu_remove_by_name` removes all items whose name field compares equal to `name`.

`ttk_menu_item_updated` updates various internal data structures for `item`. It should be called whenever you modify the item structure.

`ttk_menu_updated` restores the menu to the state it was in when it was initialized. **This includes removing all appended and inserted items.** You probably won't need to use this function very often.

`ttk_mh_sub` is the menu handler for launching a submenu.

`ttk_md_sub` creates the data structure used by `ttk_mh_sub`. This is just a cast of the passed `submenu` item list to `void *`.

5.2 Textarea

The textarea widget provides a way to scroll through a long string (maybe loaded from a file, etc). It does automatic word-wrapping with a good and fast algorithm (only one pass over the text). It allows you to scroll through the text with the scroll wheel, or move a page at a time with the `|<<` and `>>|` buttons.

Really, this is an extremely simple widget from an API point of view. The functions (not many) are described below.

```
TWidget *ttk_new_textarea_widget (int w, int h, const char *text,
                                  ttk_font font, int baselineskip);

TWindow *ttk_mh_textarea (struct ttk_menu_item *this);
```

```
void *ttk_md_textarea (char *text, int baselineskip);

/* Standard handlers, don't call except from your own.
   Not documented here. */
void ttk_textarea_draw (TWidget *this, ttk_surface srf);
int ttk_textarea_scroll (TWidget *this, int dir);
int ttk_textarea_down (TWidget *this, int button);
void ttk_textarea_free (TWidget *this);
```

`ttk_new_textarea_widget` creates and returns a new textarea widget based on the passed parameters. The widget will be `w` × `h` pixels at $(0, 0)$. The `text` will be displayed using `font` in the widget. The text will be wrapped once and never looked at again; changing it will do nothing, since the widget operates on a copy and frees it when it itself is freed. `baselineskip` is the number of pixels between two lines of text; this should be a few pixels above `ttk_text_height(font)` but its exact value is a question of style.

`ttk_md_textarea` creates a data structure for use by `ttk_mh_textarea` when some menu item is selected. You must pass the `text` and `baselineskip` values; they will eventually be passed to `ttk_new_textarea_widget`. This will set `font` to `ttk_textfont`, and `w` and `h` to the width and height of the menu where the item is selected.

`ttk_mh_textarea` is the textarea menu handler. `item->data` should be set to the pointer returned from `ttk_md_textarea`.

5.3 Slider

The slider widget is used to set an integer value, maybe a setting. It constrains the value between a specified minimum and maximum and allows the user to select a value between those two with the scroll wheel. A user-specified callback may be used to allow e.g. settings to change immediately, so the user can see what they are selecting. You may also set your own slider image, to be used instead of the default that comes from the current color scheme.

On the iPod, scroll events are amplified so that a full rotation of the wheel is equal to the distance between `min` and `max` on the slider.

The (not too many) functions of the API for this widget are described below.

```
TWidget *ttk_new_slider_widget (int x, int y, int w, int min,
                                int max, int *val, const char *title);
void ttk_slider_set_bar (TWidget *this, ttk_surface empty, ttk_surface full);
void ttk_slider_set_callback (TWidget *this, void (*cb)(int cdata, int val),
                              int cdata);

TWindow *ttk_mh_slider (struct ttk_menu_item *this);
void *ttk_md_slider (int w, int min, int max, int *val);

/* Standard handlers, don't call except from your own.
   Not documented here. */
void ttk_slider_draw (TWidget *this, ttk_surface srf);
int ttk_slider_scroll (TWidget *this, int dir);
int ttk_slider_down (TWidget *this, int button);
void ttk_slider_free (TWidget *this);
```

`ttk_new_slider_widget` creates a new slider widget `w` $\times$ `h` pixels, with its upper-left corner at (x, y). The left end of the slider will correspond to the value `min`, the right end to `max`, and the initial value will be `*val`. The `val` pointer must be valid for the lifetime of the widget, or until you call `ttk_slider_set_callback`. If you set `title` to something nonzero, it will be displayed above the slider.

`ttk_slider_set_bar` sets the slider to look like `full` when $val =$ `max` and `empty` when $val =$ `min`. At points in between, the left side of the slider will be taken from the left side of `full` and the right side will be taken from the right side of `empty`.

`ttk_slider_set_callback` causes `cb` to be called with two parameters—`cdata` and the value that was just set—whenever the value is changed. After this function is called, the original `val` pointer passed to the widget constructor is allowed to go out of scope.

`ttk_md_slider` returns a structure to be used in the `data` pointer of a `ttk_menu_item` whose handler is set to `ttk_mh_slider`. Its parameters are the same as those of the widget constructor, but `title` is omitted and set to zero.

`ttk_mh_slider` is the menu handler function that creates and returns a window containing a slider when some menu item is selected.

5.4 Image Viewer

The image viewer is one of the most powerful widgets in TTK. It allows you to view an image (PNG, JPEG, PNM, XPM, and various uncompressed formats are supported) in any of a variety of zoom levels, and scroll through it if it is bigger than your screen. On grayscale displays, it uses Floyd-Steinberg dithering to generate a good-looking picture. While the image viewer widget is active, the scroll wheel is used to scroll on either the x or the y axis, and the center button changes the axis being scrolled upon. `|<<` and `>>|` are used to change the zoom level; `>/||` selects actual size display, or the previous zoom level if the current display is actual-size. `MENU`, as always, exits the widget.

The function API of this widget consists of exactly one important function:

```
TWindow *ttk_new_imgview_widget (int w, int h, ttk_surface img);
```

This function creates a new image viewer widget, $\texttt{w} \times \texttt{h}$ pixels, with its upper-left corner at $(0, 0)$, to display `img`. The supplied image should remain valid throughout the lifetime of the widget. It will **not** be freed when the widget is freed.

To launch this widget from a menu item, the obligatory `mh`/`md` function pair is provided.

```
TWindow *ttk_mh_imgview (struct ttk_menu_item *this);
void *ttk_md_imgview (ttk_surface srf);
```

The operation of these functions should be self-explanatory by now, except for the fact that, since no width and height are provided, the created widget uses the width and height of the parent menu.

Chapter 6

Data Files

TTK requires several data files present to run properly. You must have a font list with at least one valid font in it, and a color scheme called `default` (which may be a symlink). This chapter describes the format of these files.

6.1 Fonts

TTK supports several types of font files.

PCF and Microwindows FNT are supported on all versions of TTK, for both the Nano-X and SDL backends. They can have variable foreground and background colors, but not anti-aliasing.

SFont fonts are a pair of `.png` files, one the RGB inverse of the other, named ⟨*font*⟩`.png` for the black version and ⟨*font*⟩`-i.png` for the white version. If you set the PNG background to transparent, these may have anti-aliasing and variable background colors; if not, anti-aliased fonts can only have the background they have in the file. In all cases, foreground color can only be "black" (the color of the characters in the `.png` file) and "white" (the color of the characters in the `-i.png` file).

To create an SFont font, draw with a graphics program the string

```
! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _ `
a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~
```

(all on one line, with spaces between each character; note that the last character of the second line is a backtick/grave accent). Use antialiasing if you want it, and make the text the color you want it to be when non-inverted. From there, you have two options: (1) you can find `sfontmaker` on Google and run it on the file; or (2) you can put magenta (all red and blue, no green) pixels on the top row in the spaces between characters manually. Save the image as ⟨*font*⟩`.png`. Then, invert it and save the inverted version as ⟨*font*⟩`-i.png`.

SFont fonts may only be used when TTK is compiled without `-DNO_SF`, which is not the default. They only work on the iPod photo.

TrueType fonts are supported, but they may be slow. TTK must be compiled with `-DNO_TF`.

6.2 Font List

All TTK programs use one set of fonts, which will be loaded on demand and stay loaded for the lifetime of the program. This list of fonts is defined in the *font list file*, which uses a special but simple format.

All font-related things are located in the "font directory"; this is `/usr/share/fonts` on the iPod and `./fonts/` on the desktop. The font list file is `fonts.lst` in this directory. (The name is subtly different from that used by `podzilla` because the format is subtly different as well.)

In the file, lines that start with `#` are comments. All other lines, except blank lines, should follow this format:

```
[File_Name_without_extension] (Displayed name) <SZ> {OFF}
```

The four fields may occur in any order, and `{OFF}` is optional. Each must be delimited with the specified delimiters.

`[Filename]` is the name of the font in the font directory, without the extension. The best extension that exists and is supported will be used. The search order is: SFont, TrueType, `fnt`, `pcf`. The first two will only be tested on color displays (iPod photo).

`(Displayed name)` is the name of the font that will be used in the application, and what will be expected to be passed to `ttk_get_font`[`info`]`()`.

`<SZ>` is the point size of the font. This is used **only to find the closest match when getting a font**; it has no effect on the return value from `ttk_text_height()` or anything else. If the font should be considered as matching any size, set this to 0.

`{OFF}` is a vertical offset in pixels for when the font is drawn. Some fonts' characters are specified too high or too low in their bounding boxes, and the result can look unsightly. To counteract this, you can specify a pixel offset that will be added to the y coordinate for each text-drawing operation. It does not affect the return value from `ttk_text_height()`. **You must specify the sign (+ or −).** For example, for the version I had of the Chicago font, I used `{+2}`.

6.3 Color Schemes

You already know about the color scheme API; what you do not know, and what you are about to learn, is the format of the color scheme files themselves. Color scheme files are composed of three types of lines: comments, which must start with `#`; commands, like `\def black #000000`; and specifications, like `header: fg => black, bg => white`. Blank lines are ignored.

Color scheme files should generally have extension `.cs` and be placed in the schemes directory, which is `/usr/share/schemes` on the iPod and `./schemes/` on the desktop.

Commands are all lines that start with a `\`. The *very first line* of the file should be something like

```
\name My Wonderful Color Scheme
```

where the actual name is whatever you want; this is used by `podzilla` to select a color scheme. The only other command recognized is `\def`, which has the format `\def` ⟨*name*⟩ ⟨*value*⟩. ⟨*name*⟩ should be something like `black` or `white` or `aquamarine`; ⟨*value*⟩ should be an RGB value specified in six-digit hexidecimal, like HTML: `#abcdef`.

Specifications are the meat of the color-scheme file. Here are some example specification lines that demonstrate the required format.

```
 name: key1 => val1, key2 => val2, key3.s1 => val3s1,
       key3.s2 => val3s2
       key4 => val, key5 => val
other:
       key => val
```

Specification lines have two parts: a top-level specification ("name" and "other" in this example; something like "window" or "header" or "menu" in actuality) and a list of key-value pairs. Either or both may be omitted. If present, the top-level specification must begin the line and must be followed by a colon. If not, the last encountered one is used.

The list of key-value pairs is separated by commas, and the key and the value are separated by `=>`. The key should be a string of (usually) lowercase letters; the actual name of this property will be ⟨*topspec*⟩`.`⟨*key*⟩. The key may contain dots, which are probably used for further subdivision but don't have to be.

The value is a collection of one or more things separated by spaces. There are seven types of values: solid colors, background images, spacing adjustments, gradients, corner rounding specifications, and repeatable-area and alignment specifications for images. Solid colors are specified by `#rrggbb` or a name that has been `\def`ined to that, and will result in lines, rectangles, and text being drawn with that color. Background images apply to filled rectangles only, and are of the form `@file.png`; `file.png` should be a file in the schemes directory, which you might want to prefix with the basename of your color scheme filename. They will be tiled to fit.

Spacing applies to horizontal and vertical lines and rectangles; it lets you "nudge" things in or out (rectangles), left or right (vlines), or up or down (hlines). It is specified as a required sign (+ or −) followed by a whole number. Spacing on a vertical line pushes it right that many pixels if positive, left that many if negative; on a horizontal line, positive values push the line down. For rectangles, *positive* spacing of n makes the rectangle *thinner* by $2n$ in both dimensions, with the same center; negative dimensions make it thicker. Think of spacing for rectangles as like the "shrink selection" tool in your graphics program with that many pixels as an argument.

Rounding of the corners on a rectangle is called for with $*$ (an asterisk) followed by a whole number below 10. The number specifies the radius of

the quarter-circle with which each corner will be rounded; 0 is no rounding, 1 is just a pixel taken out, and 9 is a very round corner.

Gradients are specified by enclosing two or three color specifications, separated by the word "to", in ⟨*angle brackets*⟩. For instance, `<#000000 to #aaaaaa to #ffffff` would specify a two-part grayscale gradient that changes color more rapidly for the first half than for the second. You may specify whether a gradient changes color horizontally or vertically by including the keyword `horiz` or `vert` somewhere inside the angle brackets; a vertical gradient is the default. Finally, you may specify a solid color in which a bar will be drawn on top of the gradient; this can be used to create "glassy" effects. The syntax is "`with` *color*" after the "*color* `to` *color*" specifying the gradient. To determine where the bar goes, you may specify an offset from top, right, bottom, and left by using `@T,R,B,L` where T, R, B, and L are in pixels (no unit) or in percent (suffix them with `%`). If you specify one value, it will be used for all; if you specify two, the first will be used for top/bottom and the second for left/right.

Repeated-area specifications are given by putting a token of the form `WxH+X+Y` in the value for a property; this will cause only the area of size $W \times H$ with its upper-right corner at (x, y) in the image to be repeated to stretch. The "corners" of the image will be put into the corners of a rectangle drawn with it; the "sides" will be tiled in one direction; and the "middle" will be tiled in both directions.

You can specify image alignment in both directions: horizontal and vertical. For horizontal alignment, include a token like `horiz left` or `horiz center` in the value for the property; for vertical alignment, `vert top`, `vert center`, etc.

A great deal of effort has been expended to ensure that images can be drawn correctly, scaleably, and good-lookingly. The parameters for repeated-area and alignment are very powerful, and you should play with them to get the effect you want.

A color scheme that emulates the Apple firmware is below.

```
\name Standard (B&W)
# Supplied with TTK as schemes/mono.cs.
\def black  #000000
\def white  #ffffff
\def gray   #a0a0a0
```

```
\def dkgray #505050

  header: bg => white, fg => black, line => black -1, accent => gray
 battery: border => black, bg => white, fill.normal => black +1,
          fill.low => dkgray +1, fill.charge => gray
    lock: border => black, fill => black
 loadavg: bg => gray, fg => dkgray, spike => black

  window: bg => white, fg => black, border => gray -3
  dialog: bg => white, fg => black, line => black,
          title.fg => black,
          button.bg => white, button.fg => black, button.border => gray,
  button.sel.bg => gray, button.sel.fg => black, button.sel.border => black,
          button.sel.inner => black +1
   error: bg => gray, fg => black, line => black,
          title.fg => black,
          button.bg => gray, button.fg => black, button.border => black,
          button.sel.bg => dkgray, button.sel.fg => white, button.sel.border => black,
          button.sel.inner => gray +1
  scroll: box => black, bg => white +1, bar => dkgray +2

    menu: bg => white, fg => black, choice => black, icon => black,
          selbg => black, selfg => white, selchoice => white, selicon => white
  slider: border => black, bg => white, full => black
textarea: bg => white, fg => black
```

That file demonstrates all the properties used by TTK and `podzilla`; most of them should be self-explanatory if you know that `sel` means "selected" and `fg` usually refers to the color of text.

You could make an "inverted" version simply by changing the color definitions at the top.

That's all! Have fun!

Chapter 7

Acknowledgements

TTK was written by Joshua Oreman, with input from the other members of the iPodLinux development team.