PEBBLE DEVELOPMENT

Ethan Gaebel
Virginia Tech
Why Should You Care?

• Tuesday’s lab in on Pebble Development
• You have a homework on Pebble Development
  • Must include a Pebble Watch App and Android Companion App
• Wearables seen as the next big frontier in mobile development
• People buy these things ($$$)
Types of Apps

- **Pebble Watchface**
  - Presents information such as the time, weather, and date
  - Involves minimal user interaction

- **Pebble WatchApp**
  - App for the Pebble involving some calculation based on user input

- **Pebble Companion App paired with WatchApp**
  - Pebble WatchApp that communicates with a smart device

- **Developer Console Scripting Apps**
  - WatchApp, so customizable has its own scripting language

- **All development for Pebble watches is in C (yay!)**
  - Except for a little bit of optional Javascript
Application Elements

- Pebble apps are event driven
- Developers must setup callback functions to be executed on user events
- Every main function has the same basic structure
  
  ```c
  int main() {

    init();

    app_event_loop();

    deinit();
  }
  ```

- `init()` will contain all the program setup, callbacks, UI elements etc
- `deinit()` will “tear down” things setup in `init()`, don’t leave anything out
- `app_event_loop()`, infinite loop, allows events to be picked up by listeners
Pebble API in General

• All the structs are typedef-ed
  • Instead of struct Window, we can just type Window (phew)
• Functions relating to certain structs are prefixed with the struct name
  • Ex. `window_set_window_handlers` deals with Window structs
  • Ex2. `layer_add_child(…)` deals with layer structs
  • Ex3. `menu_cell_basic_draw(…)` deals with MenuLayer structs
Pebble API in General

• The Pebble API is object oriented  
  • What? In C?
• Functions are bound to structs  
  • Structs have fields storing function pointers in the structs
• Structs of the same variety (i.e. Layer, MenuLayer, TextLayer) contain instances of their “parents”  
  • Access these fields by calling function (a getter, if you will)  
  • i.e. Layer *layer = menu_layer_get_layer(menu_layer);
Pebble API in General

• Explicit dynamic memory allocation is discouraged
  • Pebble has very limited memory
  • i.e. calls to malloc and calloc
  • To allocate and free dynamic memory, Pebble API calls should be used
    • i.e. window_create_window, window_destroy_window, layer_create_layer, menu_layer_create_layer
• Pointers…pointers everywhere…..
Function Pointer Example

• typedef void (*WindowHandler)(struct Window *window)
  • Declares function with void return value that take struct Window to be referenced by WindowHandler type

Ex.

    void my_function() {
        //stuff
    }
    int my_function2(struct Window *window) {
        //better stuff
    }
    void my_function3 (struct Window *window) {
        //best stuff
    }

WindowHandler *handler = my_function;
WindowHandler *handler2 = my_function2;
WindowHandler *handler3 = my_function3;
Function Pointer Example

- `typedef void (*WindowHandler)(struct Window *window)`
  - Declares a function with a void return value that takes a struct `Window` to be referenced by the `WindowHandler` type.

Ex.

```c
void my_function() {
    //stuff
}
```

```c
int my_function2(struct Window *window) {
    //better stuff
}
```

```c
void my_function3 (struct Window *window) {
    //best stuff
}
```

```c
WindowHandler *handler = my_function; //DOESN’T WORK
WindowHandler *handler2 = my_function2;
WindowHandler *handler3 = my_function3;
```
Function Pointer Example

- `typedef void (*WindowHandler)(struct Window *window)`
  - Declares function with void return value that take struct Window to be referenced by WindowHandler type

Ex.
```c
void my_function() {
    //stuff
}
int my_function2(struct Window *window) {
    //better stuff
}
void my_function3 (struct Window *window) {
    //best stuff
}
WindowHandler *handler = my_function; //DOESN’T WORK
WindowHandler *handler2 = my_function2; //DOESN’T WORK
WindowHandler *handler3 = my_function3;
```
Function Pointer Example

- typedef void (*WindowHandler)(struct Window *window)
  - Declares function with void return value that take struct Window to be referenced by WindowHandler type

Ex.

```c
void my_function() {
    //stuff
}
int my_function2(struct Window *window) {
    //better stuff
}
void my_function3 (struct Window *window) {
    //best stuff
}
WindowHandler *handler = my_function; //DOESN’T WORK
WindowHandler *handler2 = my_function2; //DOESN’T WORK
WindowHandler *handler3 = my_function3; //SUCCESS!
```
A Pebble Function Pointer Example

```c
static Window *window;

void window_load() {
    //do stuff to setup window like set layers
}
void window_unload() {
    //destroy elements of the window
}
void init() {
    window = create_window();
    WindowHandlers winHandle;

    winHandle.load = window_load;
    winHandle.unload = window_unload;

    window_set_window_handlers(window, winHandle);
    window_stack_push(window);
}
```
Visual Elements

• Window
  • Fundamental UI element of all pebble apps
  • Analogous to an xml layout file in Android
  • Pushed and popped onto window stack for visibility
  • One, and only one, must be displayed at all times
    • Except when animating transitions between windows
  • Handle all user input (button clicks) by using callback functions
    • These callback functions can only be set once per Window
Essential Window Functions

- Window* window_create()
  - Create new window, return a pointer to it

- void window_set_click_config_provider(Window, ClickConfigProvider)
  - Set a function with the signature void <function_name> (void *context) to run every time the window is brought into focus
  - Function passed must setup all button click handlers
    - i.e. the window_single_click_subscribe function below

- void window_single_click_subscribe(ButtonId, ClickHandler)
  - Set callback function for a single button click specified by button_id
  - i.e. BUTTON_ID_SELECT
Window Actions Setup Example

static Window *window;

static void select_handler(ClickRecognizerRef recognizer, void *context) {
    //Action to execute when select is clicked
}

static void up_handler(ClickRecognizerRef recognizer, void *context) {
    //Action to execute when up is clicked
}

static void down_handler(ClickRecognizerRef recognizer, void *context) {
    //Action to execute when down is clicked
}

static void click_config_provider(void *context) {
    window_single_click_subscribe(BUTTON_ID_SELECT, select_handler);
    window_single_click_subscribe(BUTTON_ID_UP, up_handler);
    window_single_click_subscribe(BUTTON_ID_DOWN, down_handler);
}

static void window_load(Window *window) {
    window_set_click_config_provider(window, click_config_provider);
}

int main () {
    init();
    app_event_loop();
    deinit();
}
Visual Elements

- **Window Stack**
  - Hold all currently, previously displayed windows (unless explicitly removed)
  - Top of stack is the currently displayed window
  - Simple push/pop operations to change out windows
  - Can remove windows by index from the stack (but not add)
Essential Window Stack Functions

- **void window_stack_push(Window *window, bool animated)**
  - Pushes passed in window onto top of window stack, making it visible

- **Window* window_stack_pop(bool animated)**
  - Pops the currently visible window off the window stack

- **bool window_stack_remove(Window *window, bool animated)**
  - Removes passed in window from stack, returns false on failure
  - NOTE: There is no corresponding add function
Visual Elements

• Layers
  • Display text, images, other layers
  • Many types
    • MenuLayer, ActionBarLayer, TextLayer, BitmapLayer, MenuBarLayer and more….
  • Every Layer type (TextLayer, MenuLayer etc) contains a base Layer object that provides the same fundamental operations
  • Store information about state necessary to draw or redraw the object that it represents
Layer Details

• Pass a GRect struct to layer_create, must define what space the layer will occupy
  • GRect has two fields, origin, and size

  • origin: specifies where the layer starts, is GPoint struct with two int fields (x, y)
    • NOTE: The origin of the pebble is at the top left corner of the screen

  • size: specifies size of rectangle and is GSize struct with two int fields (h and w) (height and width)
Layer Details

- Layers can store data, i.e. a callback function, by calling `layer_create_with_data` and passing size of data region.
  - Data is set by calling `layer_get_data(const Layer *layer)`.
    - Return `void*` type pointing to data and manipulating data at address.
Layer Details

• Every Layer (MenuLayer, TextLayer, BitmapLayer) contains a field of plain old Layer type
  • Provides useful properties of polymorphism
  • Allows passing around Layer reference contained in MenuLayer to a function that only accepts the Layer type
Essential Layer Functions

- `Layer* layer_create(GRect frame)`
  - Create a layer, size determined by GRect struct
- `void layer_destroy(Layer *layer)`
  - Destroy the layer
- `GRect layer_get_frame(const Layer *layer)`
  - Gets the bounds of the frame in the form of a GRect struct
- `struct Window* layer_get_window(const Layer *layer)`
  - Get Window struct layer is in or NULL if layer not bound to window
- `void layer_add_child(Layer *parent, Layer *child)`
  - Set child layer inside parent layer
  - Probably the most used layer function....
Text Layer

- Simple layer that provides functions to write and erase text
- Can set text color, font, background color, text alignment…
- Simplest Layer
Menu Layer

- Layer which defines a familiar menu layout
  - Each cell can have its data altered
- Heavy to setup, minimum of about 5 callback functions
- Little interaction required afterwards (unless you’re doing something tricky)
**Bitmap Layer**

- Used to display a picture
- Good for icons and simple figures, no HD pictures…
Action Bar Layer

• A layer which provides a vertical row of buttons on the right side of the window
  • See the default music player app on the Pebble for an example
• Can contain up to 3 customizable icons (i.e. next, prev, play)
• Icons can be swapped out in real-time
• ActionBarLayer is bound to the window directly
  • No intermediary layer
  • All click handlers are automatically setup on binding
  • Additional Layers may cover up the ActionBar
Persistence on the Pebble

- Storage space is identified by the (hopefully) unique app UUID
- Values are all stored in key, value pairs
  - Keys are uint32_t values
  - Values are integers, c-strings (char *), and byte arrays
  - structs can be saved as byte arrays too!
- Maximum storage space for any single app is 256 bytes
- Calls to Persistence API are slow
  - best used in the init() and deinit() functions
Persistence Function Calls

• Writing
  • `persist_write_bool(BOOL_KEY_VALUE, true/false);`
  • `persist_write_int(INT_KEY_VALUE, 42);`
  • `persist_write_string(STRING_KEY_VALUE, "Douglas");`
  • `uint8_t bytes[42];
    persist_write_data(BYTES_KEY_VALUE, bytes, sizeof(bytes));`

• Reading
  • `bool truth = persist_read_bool(BOOL_KEY_VALUE);`
  • `char username[20]
    persist_read_string(STRING_KEY_VALUE);`
  • `uint8_t bytes[42];
    persist_read_data(BYTES_KEY_VALUE, bytes, sizeof(bytes));`

• Existence
  • `bool exists = persist_exists(QUESTIONABLE_KEY);`
Pebble Device Communication

- Communication can be initiated from device or the Pebble
- Phone companion app must have the unique UUID of the app to communicate with it
- All data must be sent as a dictionary, in key-value pairs
- Two packages to use for communication:
  - AppMessage
  - AppSync
- Additional data structures provided on both Pebble and Android
  - PebbleDictionary
  - Tuple
  - Tuplet
Pebble Communication with AppMessage

• Allows high level of control over each individual message
• Must implement at most 4 callback functions
• Sending
  • Write values to Dictionary and call “app_message_outbox_send()”
• Receiving
  • void in_received_handler(DictionaryIterator *iter, void *context)
    • In body check for fields you are expecting to receive with:
      • dict_find(DictionaryIterator *iter, int id)

• Older firmware (1.1) doesn’t support AppMessage
Android Communication with AppMessage

• Import PEBBLE_KIT project into Eclipse and add to Build Path of Android apps

• Receiving Messages
  • registerReceivedDataHandler
  • One function to implement:
    • void receiveData(final Context, final int transactionId, final PebbleDictionary)
  • Must acknowledge receipt of message (or NACK it)
    • PebbleKit.sendAckToPebble(final Context context, final int transactionId);

• Sending Messages
  • sendDataToPebble(final Context, final UUID, final PebbleDictionary)

• Status Updates
  • Listen for watch connected event
  • Listen for ACK/NACK messages from the Pebble
Pebble Communication with AppMessage

Pebble OS

Watch App
- \texttt{app\_message\_out\_send()}
- \texttt{.out\_sent\_called}
- \texttt{app\_message\_out\_send()}
- \texttt{.out\_failed\_called}
- (accepting) \texttt{.in\_received\_called}
- (busy / no space) \texttt{.in\_dropped\_called}

Dictionary
- \texttt{“Ack”}
- \texttt{“Nack”}

Phone App
- accepted!
- rejected!
- sending message
- sending message
Pebble Communication with AppSync

- Built on top of AppMessage
- Maintains and updates a single Dictionary
- Has built-in listeners to automatically update UI elements when the Dictionary changes
- Good for applications involving many updates
  - No user-incurred synchronization costs
- Setup one callback, call a setup function, done!
Pebble Communication with AppSync

- Setup sync listeners and callbacks
- `app_sync_init`
  ```c
  struct AppSync *s,
  uint8_t *buffer,
  const uint16_t buffer_size,
  const Tuplet *const keys_and_initial_values,
  const uint8_t count,
  AppSyncTupleChangedCallback tuple_changed_callback,
  AppSyncErrorCallback error_callback,
  void *context)
  ```
- `Sync_tuple_changed_callback`
  ```c
  const uint32_t key,
  const Tuple *new_tuple,
  const Tuple *old_tuple,
  void *context)
  ```
Android Communication with AppSync

• Exactly the same as AppMessage
Pebble Communication with AppSync

Watch App
- `app_sync_init(..., current_dict, ...)`
- `tuple_changed_callback called (to init the app's user interface)`
- `app_sync_set(s, updated_tuplets, count)`
- `tuple_changed_callback called (to update the app's user interface)`
- `app_sync_set(s, updated_tuplets, count)`
- `error_callback called`
- `tuple_changed_callback called (to update the app's user interface)`
- `error_callback called`

Pebble OS
- `AppSync`
  - `update current_dict`
  - `app_message_out_send()`
  - `update current_dict`
  - `app_message_out_send()`
  - `failure, no update`
  - `update current_dict`
  - `failure, busy/no space`

Phone App
- `✓`
- `✗`
- `✓`
Javascript Aided Communication

- Platform independent way to communicate with Pebble
- Interface to make HTTP requests
  - Turns phone into a server where your Pebble is the client
- Interface from phone to Pebble using “Pebble” Javascript object
- Interface with the web using Javascript function calls
  - Part of W3C standard
- Data sent in Key-Value pairs
  - Follow JSON specification
- To make a Pebble app using Javascript
  - Create new project with command:
    `pebble new-project --javascript my_js_project`
Pebble Accelerometer

- Very sensitive
  - Sensitive enough to detect taps on the phone
- Measured in milli-Gs
  - Has a range of -4000 to 4000
- Watch vibrations affect accelerometer readings
- Grabs struct with x, y, z, bool did_vibrate indicating whether vibration occurred while grabbing values and timestamp in milliseconds
Pebble Accelerometer Axes
Using the Pebble Accelerometer

- Three main ways to utilize accelerometer
  - Register for shake or tap events
    - Predefined standards for taps and shakes
  - Process data in batch jobs to analyze for patterns
    - Can automatically poll for data at predefined intervals
  - Real time data usage
- Easy to subscribe to services for all three
JSON Configuration File

- JSON file in root directory of project (settings on CloudPebble)
- Includes various values, most are pre-generated
  - App Kind (watch app, watch face, companion app)
  - Long Name
  - Short Name
  - Menu Image
  - Version Code
  - Version Label
  - App UUID
- Also define Javascript Message Keys (if desired)
Pebble Development Setup

- Must be running Ubuntu (other Linux distros won’t work out of box)
- Download SDK and follow the instructions:
  - [https://developer.getpebble.com/2/getting-started/linux/](https://developer.getpebble.com/2/getting-started/linux/)
- There may also be some Python dependencies that are necessary to download using apt-get
- All project activities (create, build, install, etc) are issued using the “pebble” terminal utility
- To test that you have configured this correctly run:
  pebble new-project hello_world
Pebble Development

• Create a new project:
  • pebble new-project <project-name>

• Build project code:
  • pebble build (run inside the project directory)

• Install to Pebble watch:
  • Connect phone and computer to the same Wi-Fi
  • Get IP Address from Pebble watch companion app
  • pebble install --phone <ip-address of phone>

• Debug code running on Pebble:
  • To print debug messages add calls to the function below to your code
    void app_log(uint8_t log_level, const char *src_filename, int src_line_number, const char *fmt, …)
  • pebble debug --phone <ip-address of phone>
  • This will stream print statements initiated by app_log to the terminal
Uploading to the Pebble App Store

- Create various graphics to include with your app
  - To upload your Pebble app to the market you need a minimal of 4 graphics for:
    - Large Icon
    - Small Icon
    - Screenshots (at least one)
    - Header Image (at least one)
Things to Keep in Mind

• Memory is valuable, free it as soon as possible, and avoid unnecessary global variables
  • Although many global variables are necessary
• Memory is NOT managed, you must match every _create() function call with a _destroy() function call
• The interface to the Pebble is very limited…try to come up with novel ways to input data easily
Downsides

• Back button cannot yet be overridden
• Feature set still young, 2.0 SDK added persistence, accelerometer access, magnetometer and many other features
• Closed-source
• Not much memory
Need References?

• The online Pebble API is fantastic
  • https://developer.getpebble.com/2/api-reference/modules.html
• When you run pebble new-project <project_name> you get the default hello world Pebble app
• Inside the Pebble SDK folder is a folder named Examples which demonstrates most of the functionality of the Pebble watch
• PebbleCloud has several example projects you can select from