CIT 595 Spring 2017
Pebble Smartwatch Tutorial

Introduction

This tutorial takes you through the basics of creating an app for the Pebble smartwatch and for setting up the infrastructure for your group project.

Although this tutorial is not graded per se, it is a critical part of your project and you are encouraged to complete this as soon as possible.

Before you begin

Go to https://dev-portal.getpebble.com/ and create an account. You will need this in order to push apps to your watch.

Next, download and install the Pebble app (the OLD VERSION) from iTunes or Google Play on your iOS or Android phone. You should download the free one, of course! Please use the account you created to login to your Pebble phone app.

The first time you run the app, it will take you through the steps of synching your phone with your watch via Bluetooth. Make sure you're able to get them to see each other before proceeding.

The Bluetooth ID of the watch should be displayed on the watchface. If not, follow these steps to find it:

- From the watchface screen (showing the current time), click the Select button, which is the center button on the right side of the watch
- Use the Down button (under the Select button) to scroll to Settings, then click the Select button
- Bluetooth should be highlighted already; if not, use the Down button to navigate to it. Then click the Select button
- The Bluetooth ID should be listed at the top
- If the watch is already paired with another device, use the Down button to scroll to it, click Select, then click Select again to forget the pairing

Part 1: Set up environment

The first thing you will need to do is set up an environment for developing the smartwatch apps. Unfortunately, the lab machines do not have the Pebble SDK installed, so this tutorial will use an online development environment. You may choose to install the SDK locally (you can get it at https://developer.pebble.com/sdk/install/) but please do not do that until after you complete this tutorial.

Instead, go to cloudpebble.net and create a new account there.

NOTE: You will need to use Chrome or Safari in order to work with CloudPebble.

Once you have done that, create a new project named "hello world", using "Pebble C SDK" as the project type, “SDK 2” as the SDK version and "minimal" as the template.

Part 2: Hello, World

In the CloudPebble IDE, select main.c and modify it so that it looks like this:

```
#include <pebble.h>

static Window *window;
static TextLayer *hello_layer;

static void window_load(Window *window) {
  Layer *window_layer = window_get_root_layer(window);
  GRect bounds = layer_get_bounds(window_layer);

  hello_layer = text_layer_create((GRect)
  { .origin = { 0, 72 },
    .size = { bounds.size.w, 20 } });
  text_layer_set_text(hello_layer, "Hello world!");
  text_layer_set_text_alignment(hello_layer, GTextAlignmentCenter);
  layer_add_child(window_layer, text_layer_get_layer(hello_layer));
}

static void window_unload(Window *window) {
  text_layer_destroy(hello_layer);
}

static void init(void) {
  window = window_create();
  window_set_window_handlers(window, (WindowHandlers) {
    .load = window_load,
    .unload = window_unload,
  });

  const bool animated = true;
  window_stack_push(window, animated);
}

static void deinit(void) {
  window_destroy(window);
}

int main(void) {
  init();
  app_event_loop();
  deinit();
}
```

The "main" function (at the bottom) is called when the app starts. It calls "init" to create a Window, which is used to represent the user interface. When the Window is pushed onto the stack at the end of "init", that calls "window_load", which gets the Window's Layer, size it, and then sets the text to "Hello, world!"

After "init" finishes, then "main" calls "app_event_loop", which waits for user input, which we'll see in the next step. The "deinit" function is called when the user exits the program.

In the CloudPebble IDE, click "Compilation" and then click the "Run build" button. On the same screen, you should see that the build succeeded.

CloudPebble will need to know the IP address of your phone: (https://developer.pebble.com/guides/tools-and-resources/developer-connection/)

- On Android, open the Pebble application and click on the Settings menu. In the Developer Options, check Enable Developer Connection. Return on the home screen of the Pebble application, the IP address of your phone is displayed below the green button.
- On iOS, open the Pebble app → Settings (on the top left) and enable the Developer Mode option. Choose 'DEVELOPER' from the menu on the left and set the first switch to "ON". Make sure your phone is connected to the same wi-fi network as your computer to get an IP address; take note of the IP and enable the remote connection.

Now use Chrome, Safari, or Opera to push the build to your watch via your phone. Make sure your phone is connected to the same Wi-Fi network as your development machine. Choose the "PHONE" tab on the top of CloudPebble, click "INSTALL AND RUN" button.

You should see a pop-up appear in your browser indicating that your app is being pushed to the phone. Then the watch should vibrate and you should see "Hello, world!" appear in the window. Your first smartwatch app!

More information about the structure of Pebble smartwatch apps is available at https://developer.pebble.com/guides/user-interfaces/. Even though you may want to play a bit more, please try to complete all of the steps of the tutorial today during the lab time.

Part 3: Handling button clicks

Next, we will create functions that are called when the user clicks one of the buttons on the side of the watch. Add the following functions to the top of main.c, right underneath the global variables:

```
/* This is called when the select button is clicked */
void select_click_handler(ClickRecognizerRef recognizer, void *context)
{
   text_layer_set_text(hello_layer, "Selected!");
}

/* this registers the appropriate function to the appropriate button */
void config_provider(void *context) {
   window_single_click_subscribe(BUTTON_ID_SELECT,
select_click_handler);
}
```

Then add the following line to the "init" function, immediately before the declaration of the variable "animated":

```
// need this for adding the listener
window_set_click_config_provider(window, config_provider);
```

As you can see, this function call in "init" associates the "config_provider" function with the Window. The "config_provider" function associates the "select_click_handler" function with the center button (indicated by BUTTON_ID_SELECT), and that function modifies the content that is displayed in the Layer.

If you want to also detect clicks of the other buttons, you can called "window_single_click_subscribe" with BUTTON_ID_UP or BUTTON_ID_DOWN as the first argument, and then a function pointer as the second.

Now test your app by building it again and redeploying it through CloudPebble.

More information about the Pebble smartwatch UI is available at https://developer.pebble.com/guides/user-interfaces/layers/. However, as mentioned above, even though you may want to play a bit more, please try to complete all of the steps of the tutorial today during the lab time.

Part 4: Communicating with the phone

In your group project, your Pebble smartwatch will need to get data from and send commands to a server program that you write in C. However, the smartwatch is not on the Internet, but rather communicates via Bluetooth with the app that you installed on your phone. That app allows you to add extra functionality by writing a JavaScript program and including it as part of your Pebble smartwatch app.

To set up the communication with the server, we'll first look at how the watch and phone send messages to each other.

In CloudPebble, choose "Settings" and then scroll down to the entry for "PebbleKit JS Message Keys". Then set the Key Name to “hello_msg” and Key ID to “0” (both without quotes).

This tells the JavaScript program to associate the key "hello_msg" with the index 0.

Now edit main.c and add the following four functions to the top of the file, right after the global variables:

```
void out_sent_handler(DictionaryIterator *sent, void *context) {
   // outgoing message was delivered -- do nothing
 }
```

```
 void out_failed_handler(DictionaryIterator *failed,
                         AppMessageResult reason, void *context) {
   // outgoing message failed
   text_layer_set_text(hello_layer, "Error out!");
 }


 void in_received_handler(DictionaryIterator *received, void *context)
{
   // incoming message received
 }

 void in_dropped_handler(AppMessageResult reason, void *context) {
   // incoming message dropped
   text_layer_set_text(hello_layer, "Error in!");
 }
```

These are the functions that will be called when various events occur while sending messages. To register these functions, add the following lines to the "init" function, right before the declaration of the variable "animated":

```
  // for registering AppMessage handlers
  app_message_register_inbox_received(in_received_handler);
  app_message_register_inbox_dropped(in_dropped_handler);
  app_message_register_outbox_sent(out_sent_handler);
  app_message_register_outbox_failed(out_failed_handler);
  const uint32_t inbound_size = 64;
  const uint32_t outbound_size = 64;
  app_message_open(inbound_size, outbound_size);
```

Now we're ready to add the code that sends a message to the phone. Add the following to the "select_click_handler" function:

```
   DictionaryIterator *iter;
   app_message_outbox_begin(&iter);
   int key = 0;
   // send the message "hello?" to the phone, using key #0
   Tuplet value = TupletCString(key, "hello?");
   dict_write_tuplet(iter, &value);
   app_message_outbox_send();
```

Now, when you press the button on the watch, it will send a message to the phone. The most important line here is the middle line, which creates a mapping of index #0 to the value "hello?". You can also add other mappings, as well, but this is good enough for now.

Next we'll write the JavaScript code that receives the message. In CloudPebble, click the "ADD NEW" link next to APP SOURCE to create a new JavaScript file and then add the following code:

```
Pebble.addEventListener("appmessage",
  function(e) {
    if (e.payload) {
      if (e.payload.hello_msg) {
        Pebble.sendAppMessage({ "0": "Recvd: " + e.payload.hello_msg });
      }
      else Pebble.sendAppMessage({ "0": "nokey" });
    }
    else Pebble.sendAppMessage({ "0": "nopayload" });
  }
);
```

This code listens for an "appmessage" coming from the watch, and then calls the function that is specified. The variable "e.payload" includes the message that was received, and we can get the message using "e.payload.hello_msg". Note that here we use the key "hello_msg", which was mapped to index #0 (used by the C code) in our configuration above.

If the JavaScript code is able to get the value mapped to that key, it then calls Pebble.sendAppMessage, which sends a message back to the Pebble. Note that it also uses key/value pairs; here, we map the key "0" to the value that we will send back.

Last, create a global variable "static char msg[100]" in main.c, then add this code to "in_received_handler":

```
   // looks for key #0 in the incoming message
   int key = 0;
   Tuple *text_tuple = dict_find(received, key);
   if (text_tuple) {
     if (text_tuple->value) {
       // put it in this global variable
       strcpy(msg, text_tuple->value->cstring);
     }
     else strcpy(msg, "no value!");

     text_layer_set_text(hello_layer, msg);
   }
   else {
     text_layer_set_text(hello_layer, "no message!");
   }
```

This code looks for key #0 in the received message, and then makes sure that it has some value. If so, it copies it to the "msg" variable and then displays it.

When you build and then run this program, you should be able to click the button on the

watch and see the display change to "Recvd: hello?" This means that you were able to send a message from the watch to the phone, and then from the phone to the watch!

Note that the Pebble app on your iOS/Android device may need to be running (and the device needs to be awake with the Pebble app in the foreground) in order for this to work. You can find more information about communication between your watch and phone at the following:

- https://developer.pebble.com/guides/communication/
- https://developer.pebble.com/guides/communication/using-pebblekit-js/

Part 5: Communicating with a server

Now we're ready for the final step: getting the phone to talk to a server program that it connects to over the network.

First, download and compile the server.c code available in Canvas; you need to run it on a Linux or Mac machine. This will be the "server" that the phone connects to over the internet. It is very similar to the starter code that we gave you for Project #1, but when it receives a request, instead of sending back HTML, it sends a JSON object.

As you may know, JSON is a very simple format for exchanging data that essentially consists of human-readable key/value pairs. For instance, the server that you downloaded sends back a JSON object that looks like this:

```
{
"name": "cit595"
}
```

In this case, the key is "name" and the value is "cit595".

Now use CloudPebble to modify the JavaScript code in your project so that it looks like this (be sure to overwrite what was originally there):

```
Pebble.addEventListener("appmessage",
  function(e) {
    sendToServer();
  }
);

function sendToServer() {

      var req = new XMLHttpRequest();
      var ipAddress = "158.130.63.17"; // Hard coded IP address
      var port = "3001"; // Same port specified as argument to server
      var url = "http://" + ipAddress + ":" + port + "/";
      var method = "GET";
      var async = true;

      req.onload = function(e) {
                // see what came back
```

```
                var msg = "no response";
                var response = JSON.parse(req.responseText);
                if (response) {
                    if (response.name) {
                        msg = response.name;
                    }
                    else msg = "noname";
                }
                // sends message back to pebble
                Pebble.sendAppMessage({ "0": msg });
      };

        req.open(method, url, async);
        req.send(null);
}
```

This function creates an HTTP request and sends it to a server. Note that you will need to change the value of the ipAddress and port variables to refer to the machine on which your server is running; if your server is running on a lab machine, you need to use port 3001 and your phone needs to be on one of the Penn networks.

The line that begins "req.onload" defines the function that is called after the JavaScript sends the HTTP request using "req.send" and then receives a response. It looks in the response for the key called "name" (which was specified in the JSON object) and, if it exists, sends it to the Pebble using the Pebble.sendAppMessage function that we saw previously.

Build and deploy your Pebble app (you should not need to modify main.c for this) and then click the button on the watch. You should see your server indicate that it received a request, and the watch display should change to "cit595".