TECHNICAL DOCUMENTATION

Alifia Haidry, Josh Taylor, Anders Schneider

This document provides information about the functionality of our Pebble watch app. The first part (pages 1 through 5) details the messages sent between the various components of the system. The second part (pages 6 and 7) provides information about the functionality within each of the programs that comprise this project.

Message Structures

There are 6 different message interfaces in our system.

Pebble App to Phone

Messages are sent from the Pebble app to the phone using AppMessage – a bi-directional messaging system. The message is stored in a Dictionary as a key/value pair. Each type of request that the Pebble can generate is mapped to a unique key. These messages are sent either to request some information from the server or to change the state of the system in some way. These messages are generally initiated by a button push from the user (automated polling for the current temperature is one exception). For our purposes, the value portion of the message was not used.
Here is an example of the message that is sent by the Pebble to request the current temperature (key 0):

```
DictionaryIterator *iter;
app_message_outbox_begin(&iter);
Tuplet value = TupletCString(0, "Current Temp");
dict_write_tuplet(iter, &value);
app_message_outbox_send();
```

The phone has a general handler for all incoming AppMessages from the Pebble app. The message is received as a JSON object, with the message contents in the payload property. The handler checks the payload property (which is itself a JSON object) for each possible message type, until one is found. The message types are defined in the PebbleKit JS Message Keys map, where each possible integer key sent by the Pebble corresponds to a particular message type in the payload object.

Here is an example of the phone checking for the existence of a current_temp message type (key 0):

```
Pebble.addEventListener("appmessage",
  function(e) {
    if (e.payload) {
      if (e.payload.current_temp) {
        // Get current temperature
        sendToServer(0);
      }
    }
  });
```

In this example, when the phone detects a current_temp message from the Pebble, it initiates a server request for the current temperature.

Phone to Server

Messages are sent from the phone to the server in the form of an XMLHttpRequest. The IP address and port that the request should be sent to is hard-coded into the phone's sendToServer function. The key, which represents the type of request, is a parameter of sendToServer. The key is included in the message's request line as the local path of the requested information. An onerror function is executed if the XMLHttpRequest is unable to be delivered successfully.

Here is an example of how the XMLHttpRequest is set up and sent:

```
// Set up http request
var req = new XMLHttpRequest();
var ipAddress = "158.130.110.84"; // Hard coded IP address
var port = "3001"; // Same port specified as argument to server
var url = "http://" + ipAddress + ":" + port + "/" + key;
var method = "GET";
var async = true;

// Handle error in http request (e.g. Unable to connect to server)
// Just pass on error key (13) to the watch app
req.onerror = function(e) {
      // Unable to deliver request successfully
};

// Send http request
req.open(method, url, async);
req.send(null);
```

The response handler is also set up before sending the message (see Server to Phone section below for details). Here is an example of what the request looks like when it is received by the server:

```
GET /0 HTTP/1.1
```

When the server receives the request, it parses the header for the requested resource (the key).

```
// wait for connection
fd    =    accept(sock,    (struct    sockaddr    *)&client_addr,(socklen_t
*)&sin_size);

// buffer to read data into
char request[1024];

// read incoming message into buffer
int bytes_received = recv(fd,request,1024,0);

// null-terminate the string
request[bytes_received] = '\0';

// get the requested resource
char c = request[5];
int a = c - '0';
```

Depending on the value of a (the key of the requested resource), the message must be passed on to the Arduino.

Server to Arduino

When the server is turned on, a serial connection is initiated between it and the connected Arduino. If the server needs to send a message to the Arduino, for example to put it into standby mode, it uses C's serial write function. The message sent is a single byte corresponding to the key associated with the message (for example, 2 -> enter standby mode). Here is an example of how the message is sent to the Arduino:

```
if(sendMessage != -1){
	write(fd, &sendMessage, 1);
	sendMessage = -1;
}
```

sendMessage is a global variable containing the key of the message, which is set by a separate thread when a message needs to be sent. fd is the file handle for the device, which is set when the connection with the Arduino is initiated. The message is received by the Arduino using the Serial.read function. The received byte is compared against known keys and appropriate action is taken. Here is an example of how the Arduino receives the "Enter standby" message (key = 2):

```
if(Serial.available() > 0) {
	int incomingbyte = Serial.read();
	if(incomingbyte == 2){
		// Enter standby mode
		standby = true;
		danceMode = false;
	}
}
```

Arduino to Server

Serial communication is also used to send data from the Arduino to the server, using the Serial.print function. The current temperature is sent as a string, and is terminated with a newline character. If the Arduino is in standby mode or dance mode, it instead sends junk data as the string "10000.0\n". Here is an example of how the current temperature is sent:

```
if (!IsPositive)
{
	Serial.print("-");
}
Serial.print(Temperature_H, DEC);
Serial.print(".");
Serial.print(Decimal, DEC);
Serial.print("\n");
```

Temperature_H is the "integer portion" of the temperature, Decimal is the "decimal portion" of the temperature, and DEC indicates that it should be converted to a base-10 string representation of the number. The server receives the message using C's read function. It reads a series of characters from the Arduino, concatenating them onto a char buffer until it encounters the newline character, which indicates that the end of the message has been reached. If the read function returns -1, that indicates that the connection with the Arduino has been lost, and the appropriate global variable is set. Here is

an example of how the server reads characters from the Arduino and stores the result in a global variable (some unrelated implementation details are excluded):

```
while (1) {
	bzero(buf, 100)
	bytes_read = read(fd, buf, 100);
	if(bytes_read == -1){
		// lost connection to Arduino
		connectedToArduino = false;
	}
	if(bytes_read != 0) {
		// read new characters, store in local buffer
		strcat(locbuf, buf);
		if(buf[bytes_read - 1] == '\n') {
			// finished reading characters, store in global
			bzero(globbuf, 1000);
			strncpy(globbuf, locbuf, strlen(locbuf) - 1);
			bzero(locbuf);
		}
	}
}
```

The data received from the Arduino is stored globally so that it can be passed on to the phone when requested.

Server to Phone

Each XMLHttpRequest that the server receives gets a response from the server in the form of a string-representation of a JSON object. The JSON response has a single property, the name of which represents the nature of the request. In some cases the value of the property is used to pass data along (for example, the current temperature as a string), and in other cases the value is not used. Here is an example of the JSON for a current temperature request:

```
{
	"cur_temp": "24.1275000"
}
```

The JSON is sent back to the phone as a string using the send function:

```
send(fd, reply, strlen(reply), 0);
```

The JSON string is parsed by the phone using the anonymous function specified in req.onload. The JSON.parse function transforms the received string into a JSON object. The object is checked for each message type until one is found. Here is an example of how the JSON object is checked for the cur_temp message type:

```
req.onload = function(e) {
	var response = JSON.parse(req.responseText);
	if (response) {
		// Pass on response to watch app
		if (response.cur_temp) {
		    // Current temperature
		}
	}
}
```

If a recognized message type is found, the message is forwarded to the Pebble App using an AppMessage.

Phone to Pebble App

In certain cases an XMLHttpRequest sent from the phone to the server has a response that needs to be passed on to the Pebble App (for example if the current temperature data was requested). This is done using an AppMessage, which is sent from the phone using a JSON object. The JSON object has a string property that represents the key of the message type. An important note here is that the key does not necessarily need to be the same as the one specified in the PebbleKit JS Message Keys map, but for the sake of consistency we used the same keys for each message type. The value of the JSON object's property is the data that needs to be passed on to the Pebble, which is taken directly from the server's response. Here is an example of how the current temperature message (key = "0") is constructed from the server's response:

```
// Handles response from server, passes response on to watch app
req.onload = function(e) {
	var response = JSON.parse(req.responseText);
	if (response) {
		// Pass on response to watch app
		if (response.cur_temp) {
			// Current temperature
			Pebble.sendAppMessage({ "0": response.cur_temp });
		}
	}
}
```

In the event of an error connecting with the server, a key of "13" is sent back to the Pebble.

The AppMessage is received by the Pebble using the in_received_handler function. The message is received as a Dictionary which has a key, value pair stored within. The handler checks the Dictionary for each known key using the dict_find function. If the key is found, the key/value Tuple is returned. The value contains the string that represents the "contents" of the message, which is formatted by the Pebble for display. Here is an example of how the current temperature message is received by the Pebble:

```
	void in_received_handler(DictionaryIterator *received, void *context) {
		// looks for key #0 in the incoming message
		Tuple *msg_tuple = dict_find(received, 0);

		if (msg_tuple) {
			// retrieve message contents from msg_tuple
			strncpy(msg, msg_tuple->value->cstring, 6);
			msg[6] = '\0';

			// format message for display
			if (!celsius) {
				convert_to_f(msg);
				strcat(msg, " F");
			} else {
				strcat(msg, " C");
			}
			text_layer_set_text(text_layer, msg);
		}
	}
```

TECHNICAL DOCUMENTATION – NON-MESSAGE FUNCTIONALITY

PEBBLE WATCH CODE

Watch App Structure

The model underlying the app is a stack of windows, with the main window at the bottom of the stack. The main window has the simple menu layer that represents the app's main menu. Whenever the user navigates to another screen (to view the current temperature, e.g.), a new window is popped onto the stack to display the new screen. When the user presses the back button to return to the main menu, that window is popped from the stack, leaving just the original main window at the top of the stack.

The main window only ever displays a single item: the simple menu layer that represents the main menu. Other windows, which are created when the user navigates to the current temperature, statistics, or dance mode pages, display a text layer. For simplicity, our program uses a single text layer that is never destroyed; instead, it is repurposed to display the current temperature, statistics, error messages, or the dance mode screen upon request.

Number and Char Array Processing

For simplicity, all temperature data is stored in the server in Celsius and all active-time data is stored in seconds, which means the Pebble watch code does the majority of the work to create displayable information for the app. When the app is set to Fahrenheit mode, the app receives a Celsius temperature as a char array from the server and must convert it to a char array with the correct Fahrenheit temperature. In accomplishing this, we circumvented the difficulty of not being able to import normal C libraries by writing our own atof and double_to_char_array functions. In addition, the actual numerical conversion from Celsius to Fahrenheit is done in the Pebble watch code. Similarly, the watch code is responsible for taking the active time in seconds and converting it to a day-hour-minute-second format.

Polling *(Additional Feature*)

The Pebble watch code includes the functionality to repeatedly poll for the current temperature when the user has selected to see the current temperature. This is implemented using a timer that is set as soon as the user requests the current temperature for the first time. Once the designated time has elapsed, two things happen: the watch issues a new request for the current temperature and the timer resets itself for the designated time. The timer is canceled once the user presses the back button to navigate away from the current temperature screen.

SERVER CODE

Multithreaded Server

The server is implemented with two threads: one that continuously monitors the serial connection to the Arduino for incoming messages and the other that continuously waits for HTTP requests from the phone (that originate from the watch). The "serial" thread blocks until it receives a full message from the Arduino, which it processes and feeds into the statistics calculation mechanism. The "HTTP" thread blocks until it receives an HTTP

request, at which point it processes the request, accesses the desired information, and sends a response back to the phone with the appropriate contents. The two threads both access and modify the temperature statistics, which they do according to a synchronized protocol dictated by mutex locks that are required before the shared data structures can be accessed.

Statistics Calculations

The server receives a signal from the Arduino once every second. If the Arduino is not in standby or dance mode, these messages contain the current temperature – otherwise they contain junk data and simply serve to keep track of time. The server pushes these values onto the front of a queue. This queue has a defined maximum capacity and, once that capacity is reached, the queue begins to pop values from the back of the queue when each new value is pushed on the front. The server keeps track of the minimum and maximum temperature, as well as the average temperature over the last interval of time. When each new value is pushed onto (and each old value popped off of) the queue, the average temperature is recalculated and the new value is checked to determine if it is a new maximum or minimum temperature. When the minimum or maximum temperature is popped off the queue, the server does a linear scan of the queue to find the new minimum/maximum.

Active Time (*Additional Feature*)

The server monitors the time that the watch has spent in "active" mode (as opposed to "standby" mode) since the app was started. Each time the Arduino sends the current temperature, the server counts that as a second of active time. This is sent to the watch as part of the statistics display.

ARDUINO CODE

Rhythm of Data Reporting

The Arduino code serves as the metronome for our entire distributed system, as it loops continuously and sends a signal to the server (with either the current temperature or junk data) via the serial connection once per second, regardless of what mode it is in.

Temperature Reporting

The Arduino reads the numeric value of the temperature and converts it to the appropriate format to be displayed on the 7-segment display. When it receives a signal to switch into Fahrenheit, the Arduino calculates the corresponding Fahrenheit temperature and displays that value. However, it continues to send the Celsius temperature to the server, so that the server always receives consistent data.

Dance Mode (*Additional Feature*)

When the Arduino code receives the request to go into dance mode, it initiates two cycles. The first cycle is a two-state loop: the dancing man is first raising his left arm and bobbing his head left, then raising his right arm and bobbing his head right. The other cycle is the RGB color display: this is a seven-state loop that cycles through the colors red, orange, yellow, green, blue, indigo, and violet. When in dance mode, the Arduino is no longer recording the current temperature, but it still sends a signal to the server every second with junk data.