Real-time Embedded Seismic Monitoring System

Benjamin Roytburd
Department of Electrical and Computer Engineering
University of Michigan - Dearborn
Dearborn, Michigan
roytburd@umich.edu

Joel Valleroy
Department of Electrical and Computer Engineering
University of Michigan - Dearborn
Dearborn, Michigan
joelvalleroy@gmail.com

***Abstract*—We introduce a Real-time Embedded Seismic Monitoring System. Monitoring the presence of seismic activity and being able to alert the audience to it is an important safety feature used in many buildings, workplaces, industrial facilities, power plants, etc. Our System focuses on real-time requirements, ensuring the success of our goals. We also include an IoT feature to demonstrate HMI capability in showing the audience some data about the recent seismic event. Our system is also extremely affordable compared to state-of-the-art systems, while retaining most of the optional functionality.**

Keywords—Embedded, Real-Time, Seismic Monitoring

I. Introduction

Seismic monitoring systems are an important part of an early response capability for various work sites. It is important to safety for certain sites to be able to immediately know that a seismic event is happening and the details of that event. Certain machines and systems may be required to shut down if an event is detected, and depending on the event details, various maintenance or inspections may be required after.

We elected to design and build a Real-time Embedded Seismic Monitoring System. In this paper, we will discuss the details from our requirements phase to the end of our product's life cycle. Our focus is a prototype ensuring real-time functionality, affordability, and repeatability of the design so that future teams can modify it for their needs.

II. Concept

Our project involves utilizing real-time functions on a microcontroller coupled with an accelerometer, an ethernet interface, audible outputs, and interrupt options. We will build this with affordability, in contrast to other market solutions.

Seismic monitoring systems are widely used in many industries, including nuclear power plants. If a major event is detected, a reactor may automatically shut itself, or operators may choose to shut the plant down for inspection even after a non-major event. A seismic monitoring system is legally required at many such places. Seismic monitoring systems are used in other industries too, for similar reasons - equipment and structures must be inspected if a big enough earthquake is detected [1]. Therefore, this product would be intended for various customers such as: utilities, factories, seismologists, military, building engineers, and more.

Many such systems are already sold, often at exorbitant prices. These high prices are often due to nuclear quality certifications, etc. A typical solution can cost up to $30,000. These solutions often consist of the same type of design we are proposing, that is: a controller / computer, attachable sensors (accelerometers), a monitor/printer to display with.

We used the waterfall model for the life of our project, which means each distinct phase will be fully completed before we move on to the next. This will require a strict plan, and good time keeping ensuring we will be on target for the launch (live demonstration to a conference of peers). We have determined 7 distinct phases for our management timeline with the help of our textbook [2], as shown in Fig. 1.

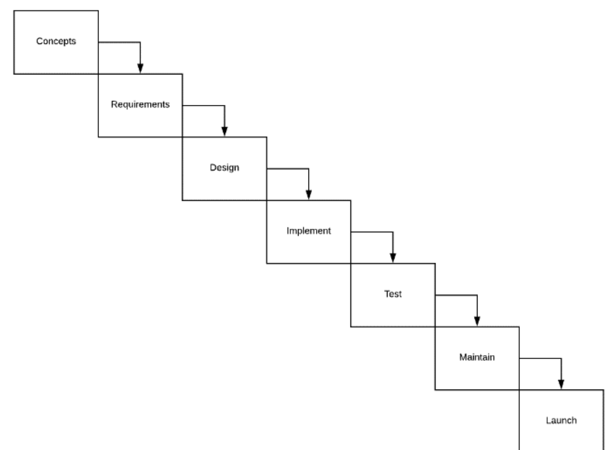

Figure 1: Software Design Life Cycle Management Process

Our project is designed to be affordable. A simple Seismic Monitoring system can be accomplished with: A microcontroller (~$20), An accelerometer ($~$10), Audio ($~1), or Video ($~20) through ethernet ($~30). This shows that creating such a device is feasible for any engineering team.

III. REQUIREMENTS

A. *Functional Requirements:*

The seismic monitor must detect and report a seismic activity in the form of accelerometer data, it must display the data in the form of a web page requested by clients and store the web page data inside its operating system. The seismic activity will constantly polled for the maximum and be able to be reset with an external interrupt, the monitor will also be able to be paused and played with external interrupts. Otherwise the monitor will report out numeric data live onto the web page. When an earthquake occurs, an external buzzer will sound to alert the user.

The seismic monitor will use an accelerometer to determine if an earthquake has occurred, and then report out the maximum acceleration data of 2 axes (x and y) in the form of meters per seconds squared onto a web page in the form of decimal numbers. The accelerometer data will be read and analyzed every 50 milliseconds. It will also be updated on a web page every 1000 milliseconds. The data should also be accurate within 10 milliseconds.

The real time operating system will also store and display the time since the last alarm, the time the current program has been running for, and the time the alarm went off on the web page.

B. *Non-functional Requirements*

The seismic monitor must run on a real time operating system (RTOS) which supports interrupts, in order to support monitoring external events such as earthquakes. The language the RTOS is built on must be high level and easy to modify / interface with, with libraries for external hardware interfacing. The code must be written without GO TO statements, minimal global variables, and minimal recursion in order to avoid long delays and lengthy code.

The RTOS must be contained on a microcontroller that has enough digital and analog I/O to support external interrupts, a networking interface, a buzzer, buttons, and an accelerometer. The microcontroller must also have a pin for voltage and ground for external modules. We will also require an affordable design.

IV. DESIGN

A. *Hardware*

The seismic monitor will use an Arduino UNO board based on the ATmega328P chipset. The ATmega328 has a modified Harvard architecture and a 8-bit RISC processor. This chip also has 32K bytes of flash memory, which should be ample space for the code for this task. The chip can communicate using serial Universal Synchronous/Asynchronous Receiver/Transmitter (USART) protocol, master/slave SPI serial interface, and an I2C interface. The board itself has 14 digital I/O pins, 6 analog input pins.

The external hardware will be a buzzer, an Arduino ethernet shield, three buttons, and an MMA8451 accelerometer. The accelerometer must output its data on at least 2-axes.

The hardware layout begins with overlaying the ethernet shield on the Arduino. The wiring shall consist of three buttons, all tied to ground with a 10 kilo-Ohm resistor each. The pause interrupt button will be tied to digital pin two, a dedicated interrupt pin on the Arduino board, and the play interrupt will be tied to digital pin five. The clear interrupt button will be tired to digital pin three, the second dedicated interrupt pin on the Arduino board. The buzzer will be tied to ground and digital pin seven. The accelerometer's SDA pin will be tied to analog pin four, and its SCL pin will be tied to the analog five pin. This is done because the accelerometer used I2C to communicate. The ground pin from the Arduino board shall be shared with the buttons, buzzer, accelerometer, and ethernet shield. The 5V power pin from the Arduino board shall be shared with the accelerometer and the ethernet shield. Reference Fig. 2 for wiring layout.

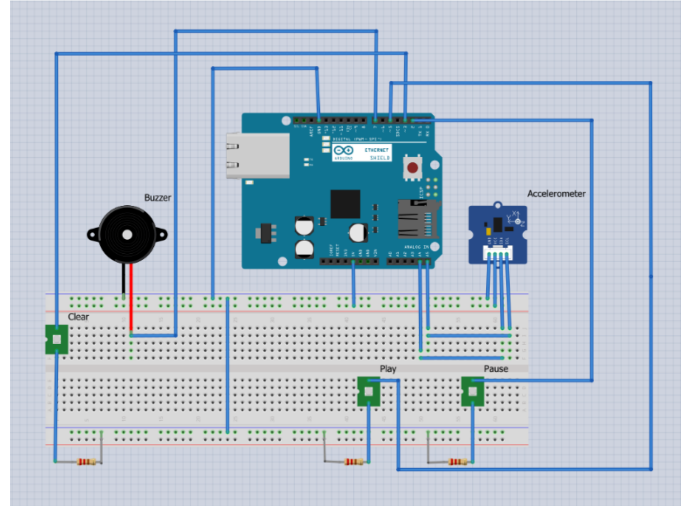

Figure 2: Hardware/Wiring Diagram

B. *Software*

The software used will be C++ and use algorithms to interpret the data from the accelerometer. The software must also store the accelerometer data as a variable that can then be displayed with HTML code also stored in the software.

The tasks shall run as a polled loop with interrupts, the code loops constantly polling the accelerometer data and if a client is available. Storing the peak acceleration will be a background task, and when an earthquake is detected, a flag is set, then the alarm will be annunciated for 500 milliseconds, the time since the program started will be stored, and the flag will be cleared. There will be two interrupts, one will be a pause button which will stop running the code until the interrupt is ended by another play button. The second interrupt will be the clear button, which will simply set the peak acceleration back to 0. In the polled loop, checking if the client is available is constantly polled, when a client requests the server, it becomes available to the server, then the server tells the client to reconnect every 1000 milliseconds, store the time since program start, and display the following on an HTML webpage: peak acceleration, time program has been running, time the alarm went off, and time since the last alarm (Figure 3. State Chart).

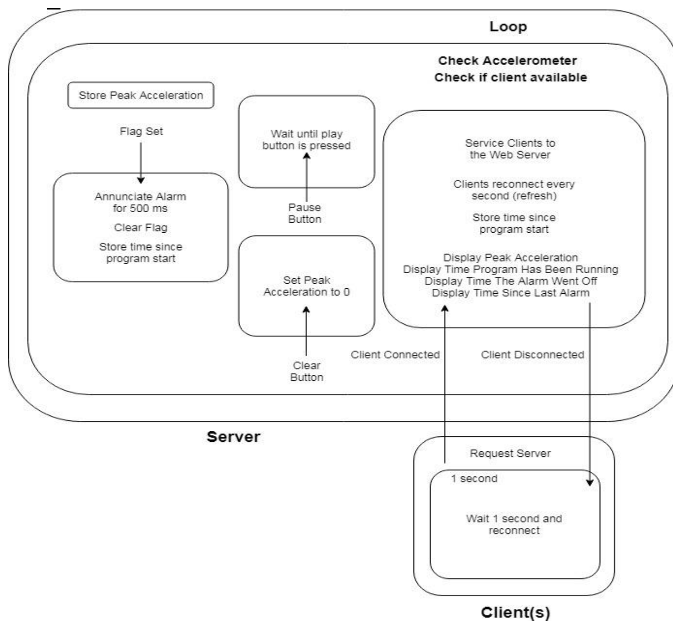

Figure 3: State Chart

The interrupts are given priority to the accelerometer check, but the web server code cannot be interrupted, it is given the highest priority. If an interrupt occurs during the web server code, it is queued and served after the web server code completes. In Fig. 4., Timing Example, an example is shown of the interrupt service routine of the real time operating system, first the accelerometer is checked, then the web server code runs, then the accelerometer check is ran again, only this time it is interrupted by the clear interrupt. Then after the interrupt finishes the web server code begins, there is an attempted clear interrupt, but since no interrupts are allowed during this time, the interrupt is serviced after the web server code completes.

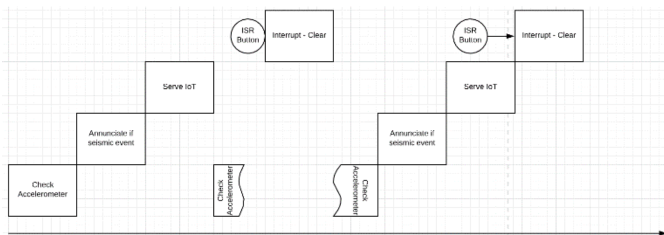

Figure 4: Timing Example

The web server is designed to work with both a local ethernet connection, as well as over Wi-Fi connected to a router. The web server must be given an IP address depending on the network and the computer that is being used. While connected, the client uses a web browser to navigate to the address of the web server. Then the client requests to connect to the web page itself, and then the server will check if the client is available while it is connected, and if so, it will transmit the web page data which will be displayed to the client in HTML. The client then refreshes its connection by disconnecting from the web server, waiting once second, then requesting to connect again (Figure 5. Networking).

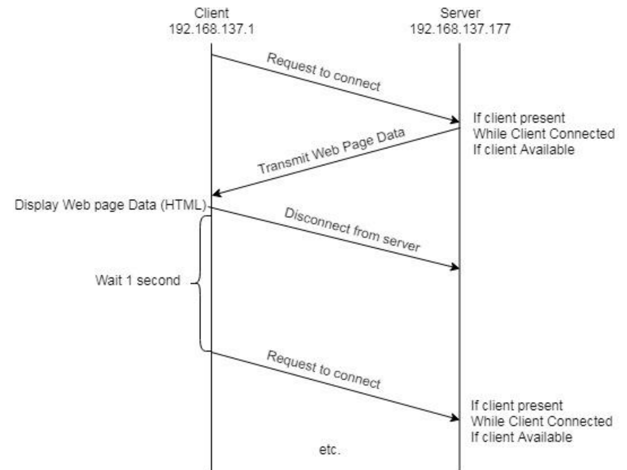

Figure 5: Networking

V. IMPLEMENTATION

Programming took place with the Arduino IDE, using its version of C++ for its microcontroller applications. To use the aforementioned hardware, the corresponding software libraries were loaded. For the accelerometer, the libraries “Adafruit_MMA8451.h”, “Adafruit_Sensor.h”, and “Wire.h” were used, and for the ethernet shield the libraries “Ethernet2.h” and “SPI.h” were used (Appendix A. Lines 6-10). The microcontroller used has a built in RTOS, which always runs the function “void setup()” first, then loops the function “void loop()” infinitely, unless otherwise specified by the user (Appendix A. Lines 21 and 39). Within “void setup()”, interrupts are attached to pin two and three, the web server is initialized, and the accelerometer is initialized (Appendix A. Lines 21-36). Within the “void loop()” function, the “eqCheck()” function is called repeatedly, this function processes the data from the accelerometer and checks if an earthquake has occurred (Appendix A. Lines 151-197). Within “void loop()”, there is also constant polling for a client, and if one is available, HTML data is sent to the client telling them to display a web page, and reconnect every second (Appendix A. 65-136). This code does not allow interrupts, since if it is interrupted it will cause the client to disconnect, and the user will have to manually reconnect the client, as opposed to doing it automatically. The interrupts are their own functions, which can be called at any time (besides during the web server code). The interrupts are “void Clear_ISR1()” and “void Pause_ISR2()”, the first interrupt resets the peak acceleration, and the second interrupt will pause the entire system until a separate play button is pressed (Appendix A. Lines 138-140 and Lines 142-149).

This code was built and tested on an assembled prototype which followed the wiring diagram (Figure 6. Physical Prototype). This code was debugged using a local PC connection, and a wireless router connection to verify the web server functionality.

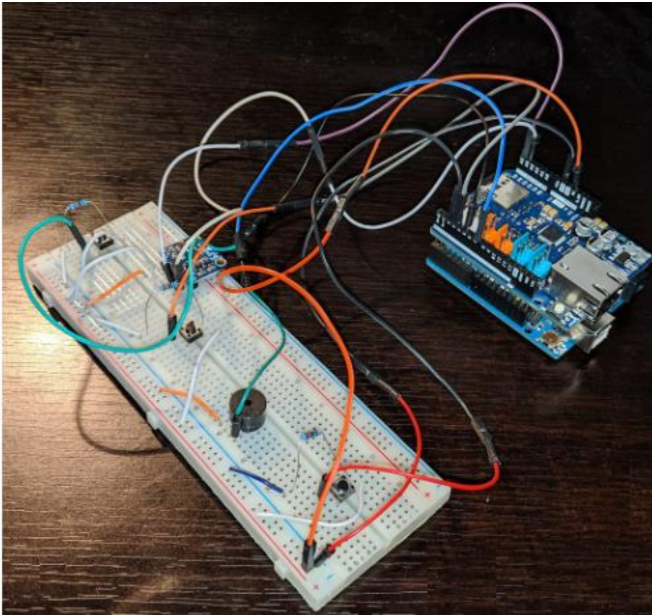

Figure 6: Physical Prototype

VI. Validation

For testing the functionality, we used system blackbox/functional testing. We verified compliance with the following requirements:

- Accelerometer is being reported correctly to the microcontroller
- Buzzer activation when magnitude threshold is passed
- Web Server HMI data matches the accelerometer/system data
- Timing meets functional requirements

We utilized standard timing functions in the Arduino standard library to allow us to time our functions. We then ran our system while logging timing data.

We found that the system takes ~5 ms to initialize and then a typical run will look like Table 1.

Functional Timing	Current time:	Time Since Last Run:
Accelerometer Check:	6 ms	
Audible Alarm:	6 ms	
Incoming Connection Check:	6 ms	
Accelerometer Check:	8 ms	2 ms
Audible Alarm:	8 ms	2 ms
Incoming Connection Check:	8 ms	2 ms
Accelerometer Check:	9 ms	1 ms
Audible Alarm:	9 ms	1 ms
Incoming Connection Check:	9 ms	1 ms
Accelerometer Check:	11 ms	2 ms
Audible Alarm:	11 ms	2 ms
Incoming Connection Check:	11 ms	2 ms
...	...	

Table 1: Functional Timing Example

As the table shows, our system is capable of staying well within its required timing limits (50 milliseconds). It's still necessary for us to keep leeway in our requirements as our HTTP server functionality can take around 30 milliseconds.

While the 'clear' interrupt is quick (~2 milliseconds), the 'pause' interrupt should only be used in testing as it causes the real-time requirements to fail.

VII. Maintenance

If we were to continue to develop this project after submission of this paper, we would switch to a different computing platform. While Arduino Uno worked great for prototyping, its memory limitations did make it difficult to accomplish more useful features on the web server. It would be wise for a future iteration of the system to be ported to a more powerful platform.

VIII. State Of The Art

State of the art seismic monitoring systems follow the same ideas we did: a computer, an accelerometer, and audio/visual interface. Some of them made for production are more robust, allowing multiple accelerometers, or redundant management computers (in our case, the Arduino). Some systems are also programmable on the user side, allowing the user to decide what magnitude of event they wish to consider for alarm. Many systems may use more advanced methods for earthquake recognition than just measuring acceleration magnitude, to allow them to make sure it's an earthquake rather than an equipment anomaly.

The visual interface for these state-of-the-art systems often include trending, and various functionality to allow the user to study the event that just happened.

A usual state of the art system purchase for an industrial application would cost at least several tens of thousands (dollars). Our system does not compete well with these competitor systems in robustness or functionality, but it does win on simplicity and cost ($~50).

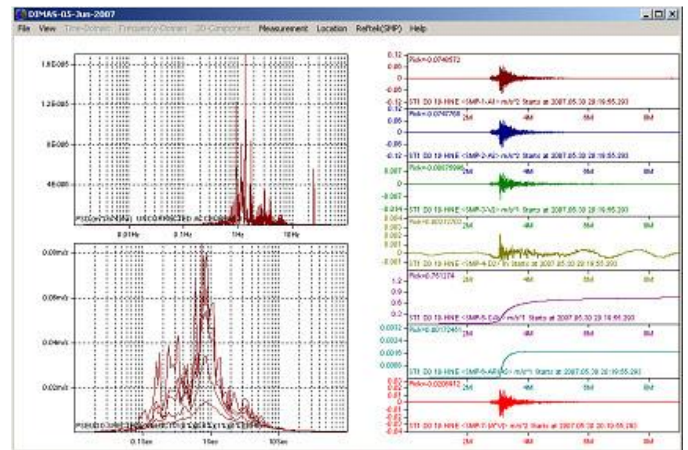

Figure 7: REFTEK RTI Software Suite [5]

IX. CONCLUSION

In conclusion, we have presented the design of our prototype Real-Time Seismic Monitoring System. The focus on real-time requirements for important functions ensures the device focuses on what is important, safety. It is vital that we separate these required functions from the frills of the system. We look forward to future uses and improvements of this technology.

APPENDIX

Appendix A. Source Code

ACKNOWLEDGMENT

We would like to thank Professor Adnan Shaout (University of Michigan – Dearborn, Department of Electrical and Computer Engineering) for his advice and guidance throughout this project. We would also like to thank the Arduino community for technical guidance on interfacing our system.

REFERENCES

[1] Skolnik, Ciudad-Real, Graf, Sinclair, Swanson, and Goings, "Recent Experience from Buildings Equipped with Seismic Monitoring Systems for Enhanced Post-Earthquake Inspection." [Online]. Available: https://www.iitk.ac.in/nicee/wcee/article/WCEE2012_5314.pdf.

[2] Laplante, P. and Ovaska, S. (2012). *Real-time systems design and analysis*. 4th ed.

[3] Learn.adafruit.com. (2019). Arduino Code | Adafruit MMA8451 Accelerometer Breakout | Adafruit Learning System. [online] Available at: https://learn.adafruit.com/adafruit-mma8451-accelerometer-breakout/wiring-and-test [Accessed 21 Apr. 2019].

[4] Arduino.cc. (2019). Arduino - WebServer. [online] Available at: https://www.arduino.cc/en/Tutorial/WebServer [Accessed 21 Apr. 2019].

[5] "RTI Software Suite for Seismic Monitoring," *Ref Tek*. [Online]. Available: https://www.reftek.com/category/products/software/rti-software-suite/. [Accessed: 22-Apr-2019].

Appendix A.

```
  1. // external third party libraries
  2. // We also use vendor code for interfacing with hardware such as :
  3. // https://www.arduino.cc/en/Tutorial/WebServer (Ethernet Hardware)
  4. //https://learn.adafruit.com/adafruit-mma8451-accelerometer-
breakout/wiring-and-test (Accelerometer)
  5.
  6. #include <SPI.h> //Arduino Standard Library
  7. #include <Ethernet2.h> //provided by Ethernet Shield Vendor
  8. #include <Wire.h> //Arduino Standard Library
  9. #include <Adafruit_MMA8451.h> //provided by Accelerometer Vendor
 10. #include <Adafruit_Sensor.h> //provided by Accelerometer Vendor
 11.
 12. Adafruit_MMA8451 mma = Adafruit_MMA8451(); //accelerometer object
 13.
 14. byte mac[] = {0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED};
 15. IPAddress ip(192, 168, 137, 177); // varies depending on your network
 16. EthernetServer server(80);
 17.
 18. double peakAcc = 0; // variable used to store peak Acceleration
 19. int eqDataIndex = 0; // tracking index for data array of seismic data
 20.
 21. void setup() {
 22.   //"pause" button is attached to pin 2
 23.   attachInterrupt(digitalPinToInterrupt(2), Pause_ISR2, HIGH);
 24.   //"clear" button is attached to pin 3
 25.   attachInterrupt(digitalPinToInterrupt(3), Clear_ISR1, HIGH);
 26.
 27.   // Ethernet Shield Pin
 28.   Ethernet.init(10);
 29.
 30.   mma.begin(); //accelerometer object
 31.   mma.setRange(MMA8451_RANGE_2_G); // sets range to 2 g
 32.
 33.   Ethernet.begin(mac, ip);
 34.
 35.   server.begin();
 36. }
 37.
 38. // Arduino loop() is a infinite FOR inside an implied C++ main() func.
 39. void loop() {
 40.
 41.   //time since last alarm
 42.   unsigned long alarmTime;
 43.   unsigned long currentTime;
 44.   unsigned long time_since_last_alarm;
 45.
 46.   int arraySize = 100; //size of ring buffers for holding seismic data
 47.   double eqDataX [arraySize]; //array for holding seismic data on X
 48.   double eqDataY [arraySize]; //array for holding seismic data on Y
 49.
 50.   bool eqFlag = false; //boolean flag, set to true if EQ detected
```

```
 51.
 52.   // function to check if accelerometer data surpasses a setpoint
 53.   eqCheck(eqDataX, eqDataY, eqDataIndex, eqFlag);
 54.
 55.   if (eqFlag){
 56.     tone(7, 500, 500); //beep at 500 Hz. for 500 ms (on pin 7 )
 57.     eqFlag = false; //reset flag
 58.     }
 59.
 60.
 61.   if (eqDataIndex > (arraySize - 1)){ //reset the index for eqData
 62.     eqDataIndex = 0;
 63.     }
 64.
 65.   //WebServer Begin
 66.   noInterrupts(); //volatile code
 67.   // listen for incoming clients
 68.   EthernetClient client = server.available();
 69.
 70.   if (client) {
 71.
 72.     // an http request ends with a blank line
 73.     bool currentLineIsBlank = true;
 74.     while (client.connected()) {
 75.       if (client.available()) {
 76.         char c = client.read();
 77.
 78.         if (c == '\n' && currentLineIsBlank) {
 79.
 80.           //https://www.arduino.cc/en/Tutorial/WebServer
 81.           client.println("HTTP/1.1 200 OK");
 82.           client.println("Content-Type: text/html");
 83.           client.println("Connection: close");  //close after complete
 84.           client.println("Refresh: 1");  // refresh the page
 85.           client.println();
 86.           client.println("<!DOCTYPE HTML>");
 87.           client.println("<html>");
 88.
 89.           //display data
 90.           client.print("<center>");
 91.           client.print("Peak acceleration of X and Y is: ");
 92.           client.print(peakAcc);
 93.           client.print("</center>");
 94.           client.println("<br />");
 95.
 96.           currentTime = millis(); //calculate time since last alarm
 97.           time_since_last_alarm = currentTime - alarmTime;
 98.           client.print("<center>");
 99.           client.print("The program has been running ");
100.           client.print(currentTime);
101.           client.print(" milliseconds");
102.           client.print("</center>");
103.           client.println("<br />");
```

```
104.
105.            client.print("<center>");
106.            client.print("The alarm went off at ");
107.            client.print(alarmTime);
108.            client.print(" milliseconds");
109.            client.print("</center>");
110.            client.println("<br />");
111.
112.            client.print("<center>");
113.            client.print("It has been ");
114.            client.print(time_since_last_alarm);
115.            client.print(" milliseconds since the last alarm");
116.            client.print("</center>");
117.            client.println("<br />");
118.
119.            client.println("</html>"); //end of processing / html
120.            break; //out of while loop
121.          }
122.
123.          if (c == '\n') { // you're starting a new line
124.            currentLineIsBlank = true;
125.          }
126.          else if (c != '\r') { // character on current line
127.            currentLineIsBlank = false;
128.          }
129.        }
130.      }
131.      delay(1);
132.      client.stop(); // close connection
133.    }
134.    //non volatile code
135.    interrupts();
136.  }
137.
138.  void Clear_ISR1(){
139.    peakAcc = 0; //resets peak acceleration
140.  }
141.
142.  void Pause_ISR2() {
143.    //pin 5 is the unpause button
144.    while (digitalRead(5) == LOW) {
145.      if (digitalRead(5) == HIGH){
146.        //play
147.      }
148.    }
149.  }
150.
151.  void eqCheck(double eqDataX[], double eqDataY[], int &eqDataIndex, bool
&eqFlag){
152.    //this function will query accelerometer and put data into memory
153.    sensors_event_t event;
154.    mma.getEvent(&event);
155.
```

```
156.   eqDataX[eqDataIndex] = event.acceleration.x; // acc. data into memory
157.   eqDataY[eqDataIndex] = event.acceleration.y;
158.
159.   eqDataIndex++; // increment index for rotating buffer
160.
161.   double peakY = 0;
162.   double peakX = 0;
163.
164.   if (eqDataX[eqDataIndex] > 0) // Don't care about negative acceleration
165.     peakX = eqDataX[eqDataIndex];
166.   else if (eqDataX[eqDataIndex] < 0)
167.     peakX = eqDataX[eqDataIndex] * -1.00;
168.
169.   if (eqDataY[eqDataIndex] > 0)
170.     peakY = eqDataY[eqDataIndex];
171.   else if (eqDataY[eqDataIndex] < 0)
172.     peakY = eqDataY[eqDataIndex] * -1.00;
173.
174.   if (peakX < 1000.00 and peakY < 1000.00){ // ignores accelerometer fault
175.     if (peakX > peakAcc)
176.     {
177.       peakAcc = peakX;
178.     }
179.     if (peakY > peakAcc)
180.     {
181.       peakAcc = peakY;
182.     }
183.
184.     peakAcc = peakAcc * 100.0; // this section truncates out some decimals
185.     int temp = (int) peakAcc;
186.     peakAcc = (double) temp / 100.0;
187.
188.     if (peakX > 1.00) //earthquake detected, Boolean flag
189.       eqFlag = true;
190.     else if (peakY > 1.00) //earthquake detected!
191.       eqFlag = true;
192.   }
193.   else {
194.     peakX = 0;
195.     peakY = 0;
196.   }
197. }
```