Model Railway Animation: Part 5, Parts & I^{2}C, Expanded
By David King

Thanks for joining while we build a clock for your layout that can display either the real time or a fast time. Both of these can be practical around our layout since there may be time that displaying the real time may be the best choice. I myself would display the real time when I open my layout for visitors to come and view the trains. I might even let some of the people (kids) run a few trains as well. Displaying the real time will help your guests and yourself not lose track of time.

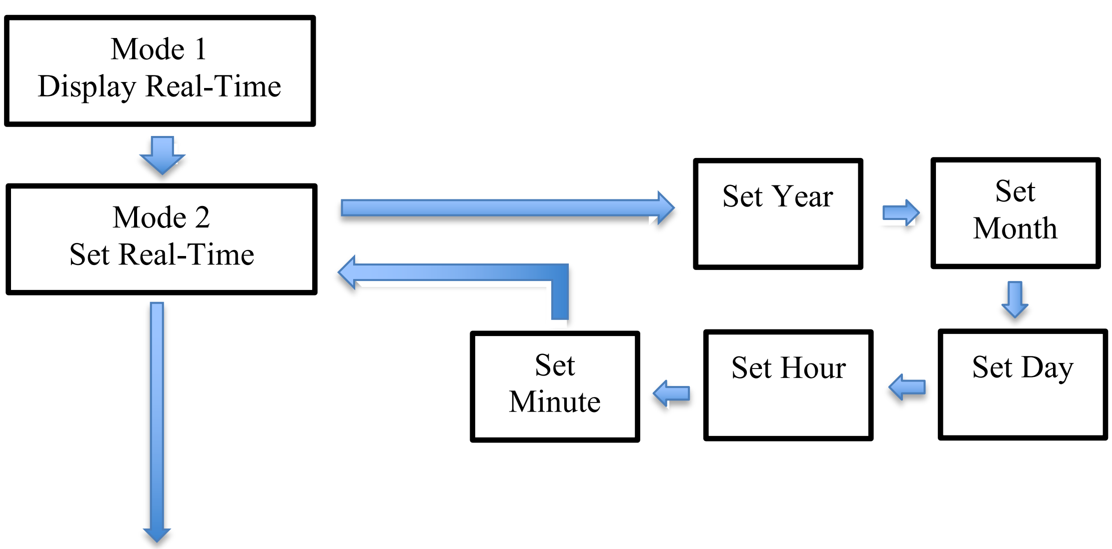
As for displaying the fast time this can be most useful if you wish to have operators run your trains on a schedule or just to simulate a more realistic pace of operations on your layout. I also find a fast clock helpful for determining how long jobs will take. For me it is nice to know how long it takes to move a set of loaded log cars from the tree loading camp in the mountains to the unloading area at the saw mill while following all of the rules operation on my layout. With this information I can create a better and more realistic operation schedule for my operators.

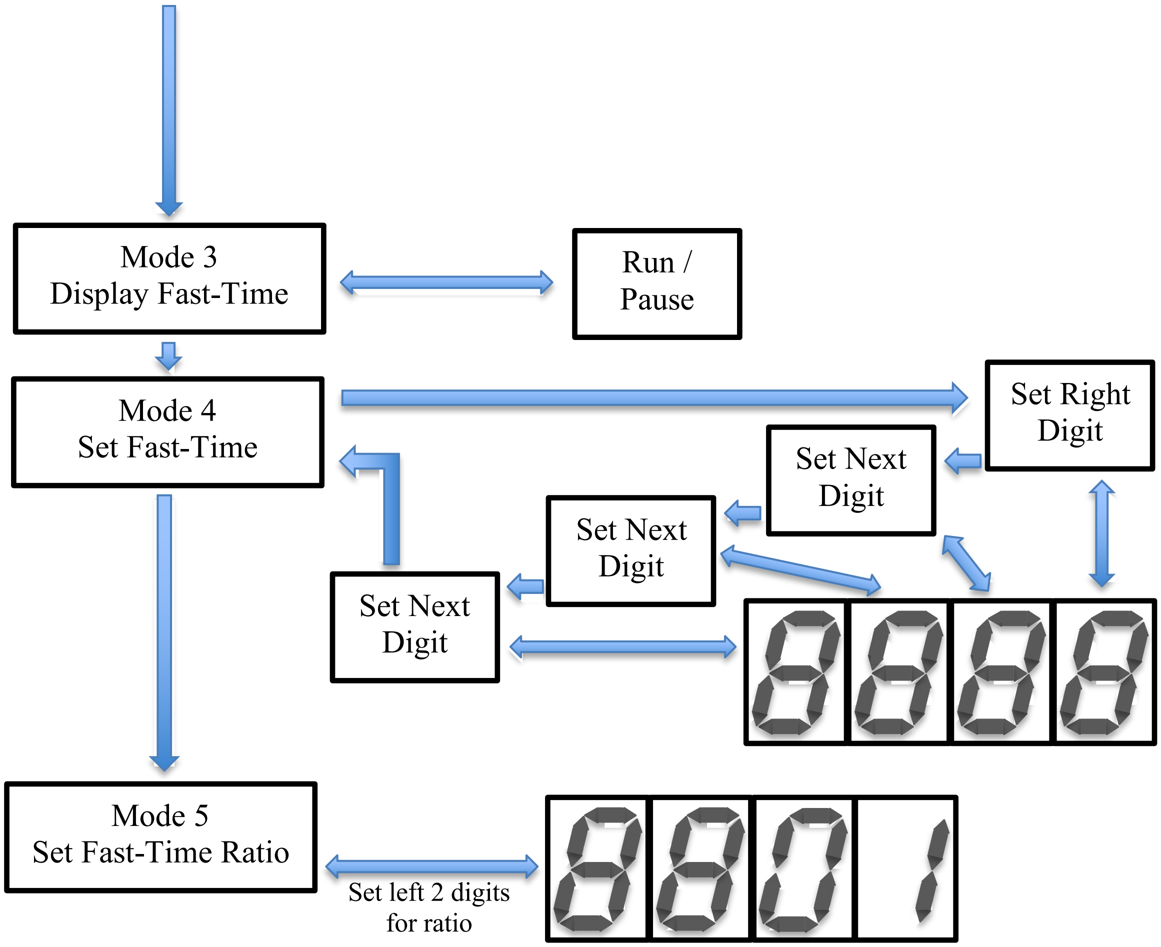
Making a Plan for the Project

In the past I've found it helpful to come up a list of requirements, wish list you might say, for the objectives. In this case I'll get us started.

- Display the Real-Time
- Display the Fast-Time
- Be able to adjust the time for both Real-Time and Fast-Time
- Adjust the ratio setting for the Fast-Time
- Be able to Run and Pause the Fast-Time

Next, we should come up with a priority or structure as to what order to be able to access or complete each of the above items on the wish list. I like to create a simplified flow chart to create this structure.

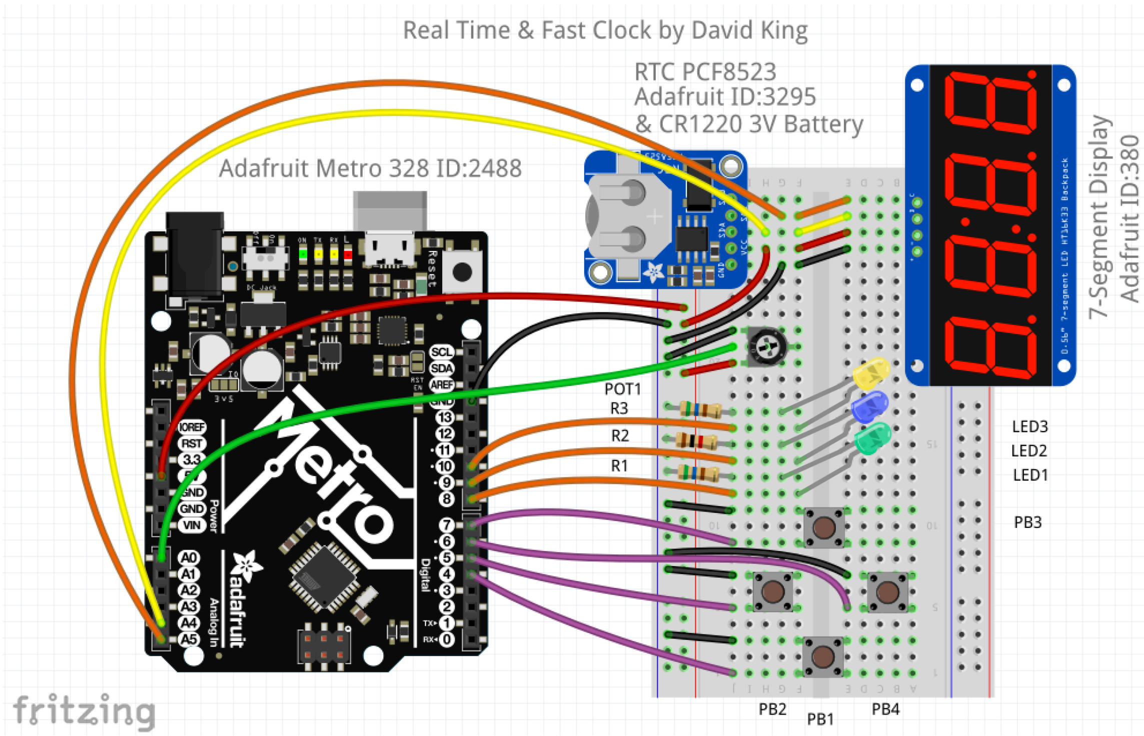




As you have most likely noticed there are many steps to the process and we will need to figure out what components are going to be used in order for our project to work. So, let us make this list now.

Component	Details/Notes	Qty.
Arduino Uno R3	Any brand	1
7 Segment Display	Adafruit Product ID: 878 for red Adafruit Product ID: 881 for blue Adafruit Product ID: 880 for green Adafruit Product ID: 879 for yellow	1
Real Time Clock Module	Adafruit Product ID: 3295	1
CR1220 3V Battery	Adafruit Product ID: 380	1
10K Potentiometer	Any supplier, POT1	1
3mm LEDS	Any supplier, 1 each red, blue, yellow	3
1k ohm resistor ¼ watt	Any supplier, R2	1
560 ohm resistor ¼ watt	Any supplier, R1, R3	2
Momentary Push Buttons	Any supplier, PB1, PB2, PB3, PB4	4
MB120 ½ breadboard	Any supplier	1
Various jumper wires	Any colours	many

Next, I have included the wiring diagram for connecting everything together. I used Fritzing to create the drawing just as I have in previous articles.



Wired up on the board we can see all of the components that I listed earlier. The Fritzing program is very useful for figuring out where to place all of the items on the breadboard and how to wire them together. Many times, I will wire up a project and then use items from this project for another project but, when I get back to the project I can use this image to re-connect everything properly together again.

You should also notice that there are a few components in the image that I did mention in the parts listing but not in the simplified flow chart showing our operational goals. These are the potentiometer, the LEDs, the resistors and the pushbuttons.

The potentiometer will be used to adjust the brightness of the clock display. The three LEDs along with their resistors will give us feedback to let us know what mode the clock is in and lastly the pushbuttons will be used to allow us to enter information into the clock. These inputs will allow us to change modes, adjust values up and down, and save/advance to the next steps.

To assist you with understanding the features of this project you can check out the YouTube link here, https://youtu.be/o1HGE2GpEsE, or on the CARM website where you found this expanded article.

Writing the Sketch

The sketch for this project is much larger than any that I have covered with you in any of the previous articles. My fully operational code is almost 700 rungs in length. This may sound daunting, but I will break it into smaller blocks of code and explain what each is used for. The biggest challenge will be taking your time with it and working your way through it with my help. Let's get started.

From the Top

The code at the top of the sketch has a few lines describing the sketch along with other information. In addition, all of the required libraries, 5 in all, are added to the sketch. Lastly, I include identifying names for the Real Time Clock module, the 7-Segment Display and commented out line to add a second display. More on the addition of additional displays at the end of the article.

Fast_Clock_CARM_01

```
// Fast_Clock_CARM_01.ino created by David King, August 2018
// Portions of the code is from Adafruit Indusries
// This code is open source and only for personal use, NOT commercial use.
// Date and time functions using a PCF8523 RTC connected via I2C, 0x68
// Time is being displayed on a 7-Segment display connected via I2C, 0x70
// Additional displays need to be unique addresses from 0x71 to 0x77
#include <Wire.h>
#include <RTClib.h>
#include <Adafruit_GFX.h>
#include <Adafruit_LEDBackpack.h>
#include <EEPROM.h>

Adafruit_7segment matrix = Adafruit_7segment();   // Name the Main Display
// Adafruit_7segment matrix1 = Adafruit_7segment();  // Name Additional Display if used

RTC_PCF8523 rtc;      // Name the Real-Time Clock breakout module

```

Next, we need to declare all of the variables needed in the sketch. For simplicity I have group variables together as to their functional use in the overall project. The first group of variables are used for the potentiometer and the pin assignments on the Uno microcontroller board. Rung 18 is the storage variable for the value from the potentiometer. Rung 19 is the analog pin used for the potentiometer. Rungs 20 to 26 are all of the digital pins used for the LEDs and pushbuttons.

```
18 int potValue;         // Pot and Pin assignments
19 int potPin = A0;
20 int greenPin = 8;
21 int bluePin = 9;
22 int yellowPin = 10;
23 int modePin = 4;
24 int runPin = 5;
25 int pausePin = 6;
26 int enterPin = 7;
27
```

Rungs 28 to 31 are used when reading the current state of the 4 pushbuttons. Rungs 32 to 35 store the values associated with each of the pushbuttons. Rungs 36 to 38 are used for the variables required for the Ratio function of the Fast-Time. Rung 36 is the Ratio value displayed on the 7-Segment Display, rung 37 is the Ratio value stored in memory and rung 38 is the memory location.

```
28 boolean modeState = true;     // PB States and Presets
29 boolean runState = true;
30 boolean pauseState = true;
31 boolean enterState = true;
32 int modeValue = 1;
33 int runValue = 1;
34 int pauseValue = 1;
35 int enterValue = 1;
36 int ratioValue;
37 int rVal;
38 int rAddress = 0;
39
```

This is the longest block of variables needed. Rungs 40 to 44 are used for the current Fast-Time values. Rungs 45 to 49 are the values that can be adjusted for the Fast-Time. Rungs 50 to 54 are the memory locations for the Fast-Time. Rung 55 is used to store the run/pause state of the Fast-Time. Rungs 56 to 68 are used to step through each setting of the Fast-Time and Real-Time. Rungs 69 to 73 are used for the adjustment of the Real-Time Clock.

```
40 int hoursL;           // Real & Fast Time variables
41 int hoursR;
42 int minutesL;
43 int minutesR;
44 int seconds;
45 int fthoursL;
46 int fthoursR;
47 int ftminutesL;
48 int ftminutesR;
49 int ftseconds;
50 int fthLAddress = 4;
51 int fthRAddress = 8;
52 int ftmLAddress = 12;
53 int ftmRAddress = 16;
54 int ftsecondsAddress = 20;
55 boolean ftRunning = false;
56 boolean setFastTimeRead = false;
57 boolean setminutesR = false;
58 boolean setminutesL = false;
59 boolean sethoursR = false;
60 boolean sethoursL = false;
61 boolean setColon = true;
62 boolean setRealTimeRead = false;
63 boolean setRealTimeWrite = false;
64 boolean setYear = false;
65 boolean setMonth = false;
66 boolean setDay = false;
67 boolean setHour = false;
68 boolean setMinute = false;
69 int wasYear;
70 int wasMonth;
71 int wasDay;
72 int wasHour;
73 int wasMinute;
74
```

This is the shortest list of variables that are required. Rungs 75 and 76 store the current time and pulse time in milliseconds from the on-board Uno timer. Rung 77 is the amount of time between pulses, the current set is 1000 milliseconds which equals 1 second. Rung 78 is used to let us know when the pulse happens.

```
75 long int timeNow;     // One second pulse variables
76 long int timePulse;
77 long int timeGoal = 1000;
78 boolean pulseTrue = false;
79
```

This is all of the variables in use for this sketch.

The Setup Code

Just like all of the variable declarations I'm going to take the setup void() and break it into smaller more manageable blocks of code. First is the pin assignment using the pinMode() functions. Here the LEDs are set as outputs and the pins for the 4 pushbuttons are set as input_pullup. Doing this with the inputs sets all of the input to have 5 volts on the pins so all we need to do is ground each of the inputs using the pushbuttons to tell when they have been activated. On rung 89 I start the 7-Segment Display by using its' name, matrix, along with the begin(). The I^{2}C address of the display is inserted between the brackets to complete this function.

Rung 90 is commented out right now but you would need this instruction if more than one display is being used. Rungs 91 to 97 are used to have the display go to the initial display of being totally blank. This starts the display with no data displayed. You would need to repeat rungs 91 to 97 if additional displays are used.

Let me take this opportunity to explain how rungs 91 to 97 work together. Rung 91 just sets the brightness of the display. The range for brightness is from 0 to 15. We will overwrite this setting in the void loop(). Rungs 92 to 96 are what will be displayed for each digit of the 7-Segment Display. The digits are numbered by position, 0 through 4 starting from the left digit. The colon in the middle is actually position number 2. The empty space, " ", and the false all mean to display nothing.

```
80 // ------------------------------ Setup Loop
81 void setup () {
82   pinMode(greenPin, OUTPUT);
83   pinMode(bluePin, OUTPUT);
84   pinMode(yellowPin, OUTPUT);
85   pinMode(modePin, INPUT_PULLUP);
86   pinMode(runPin, INPUT_PULLUP);
87   pinMode(pausePin, INPUT_PULLUP);
88   pinMode(enterPin, INPUT_PULLUP);
89   matrix.begin(0x70);
90   // matrix1.begin(0x71);       // Additional Display if used
91   matrix.setBrightness(15);
92   matrix.writeDigitNum(0, " ");
93   matrix.writeDigitNum(1, " ");
94   matrix.drawColon(false);
95   matrix.writeDigitNum(3, " ");
96   matrix.writeDigitNum(4, " ");
97   matrix.writeDisplay();
98
```

Located on rung 99 is the function to begin the Serial Monitor which allows us to see what is happening in the sketch when we have it connected to a computer.

Rungs 100 to 103 are used to talk to the Real-Time Clock module and if it can't find the module it sends a message to the Serial Monitor to inform the user the that RTC module has not been found. Rungs 105 to 112 are used a little differently in that it checks to see if the RTC module is running. If it thinks the clock has stopped or hasn't been running in the background it sends a message to the Serial Monitor. Also, if the clock has not been set it will set the date and time using your computers date and time as the current date and time. This is only done when the sketch is being installed on the Uno.

As a side note it is important to know that a battery must be installed in the RTC module for it to function. It is even possible for it to work if the battery is dead and the only down side is that RTC won't keep running when the power to the Uno is disconnected. The RTC only uses the battery when it is not being supplied by an external power supply as in power from the Uno.

```
 99  Serial.begin(57600);          // Start up serial monitor
100  if (! rtc.begin()) {          // Check for RTC being on line
101    Serial.println("Couldn't find RTC");
102    while (1);
103  }
104
105  if (! rtc.initialized()) {    // Set time in RTC if not set
106    Serial.println("RTC is NOT running!");
107    // following line sets the RTC to the date & time this sketch was compiled
108    rtc.adjust(DateTime(F(__DATE__), F(__TIME__)));
109    // This line sets the RTC with an explicit date & time, for example to set
110    // January 21, 2014 at 3am you would call:
111    // rtc.adjust(DateTime(2014, 1, 21, 3, 0, 0));
112  }
113
```

The next block of code in rungs 114 to 120 in the void setup() is used for the Fast-Time Ratio value. During start up the ratio is read from the eeprom memory on the Uno and stored in the variable ratioValue. If the value in memory is 0 as it would be the very first time you upload the sketch onto the Uno it places a 4 in the memory location and then uses that value until you set and save a different ratio value. Rungs 121 and 122 sends the message and value or rVal to the Serial Monitor for the user to see what is happening.

```
114  rVal = EEPROM.read(rAddress);     // Read the ratio from eeprom memory
115  if (rVal <= 0) {
116    rVal = 4;                       // Set default ratio if no ratio stored
117    EEPROM.write(rAddress, rVal);   // Store value in eeprom memory
118  }
119  ratioValue = rVal;              // Use the valid ratio
120
121  Serial.print("Read rVal = ");
122  Serial.println(rVal);
123
```

The next block of code is used for the retrieval and storage of the Fast-Time from the eeprom memory on the Uno. In each group of 6 rungs the values are first read in from memory and if the value is not in an acceptable rage of values for that digit a default value of 0 is substituted and written into the memory location. Doing this will ensure that the Fast-Time value will make sense. As an example, let's look at rungs 124 to 129. The left most digit of the time is the 10's of hours in a day so the only valid digits that can be in that location could be either 0, 1 or 2. Rung 125 checks to see if the number read from memory falls within the valid range. If the value read is outside of the range the code on rung 126 is used to set the value to 0 and rung 127 stores the 0 into the eeprom memory. Rung 129 then copies the value that was in memory and places it in the value used by the display.

The process of reading the values from memory, checking that the value is valid, writing a default value to memory if needed and copying the value to a variable used by the display is repeated for the rest of the Fast-Time variables.

```
124  fthoursL = EEPROM.read(fthLAddress);    // Read & Set Fast-Time values
125  if ((fthoursL < 0) || (fthoursL > 2)) {
126    fthoursL = 0;
127    EEPROM.write(fthLAddress, fthoursL);
128  }
129  hoursL = fthoursL;
130
131  fthoursR = EEPROM.read(fthRAddress);
132  if ((fthoursR < 0) || (fthoursR > 9)) {
133    fthoursR = 0;
134    EEPROM.write(fthRAddress, fthoursR);
135  }
136  hoursR = fthoursR;
137
138  ftminutesL = EEPROM.read(ftmLAddress);
139  if ((ftminutesL < 0) || (ftminutesL > 5)) {
140    ftminutesL = 0;
141    EEPROM.write(ftmLAddress, ftminutesL);
142  }
143  minutesL = ftminutesL;
144
145  ftminutesR = EEPROM.read(ftmRAddress);
146  if ((ftminutesR < 0) || (ftminutesR > 9)) {
147    ftminutesR = 0;
148    EEPROM.write(ftmRAddress, ftminutesR);
149  }
150  minutesR = ftminutesR;
151
152  ftseconds = EEPROM.read(ftsecondsAddress);
153  if ((ftseconds < 0) || (ftseconds > 59)) {
154    ftseconds = 0;
155    EEPROM.write(ftsecondsAddress, ftseconds);
156  }
157  seconds = ftseconds;
158
```

The only block of code remaining in the void setup() is used to assist us in seeing what is happening inside of the sketch. Rungs 159 to 170 are used to give us a nice display of the current Fast-Time setting as a complete time value rather than a set of individual numbers. Of course this is sent to the Serial Monitor.

```
  Serial.print("Fast-Time = ");
  Serial.print(hoursL);
  Serial.print(hoursR);
  Serial.print(":");
  Serial.print(minutesL);
  Serial.print(minutesR);
  Serial.print(":");
  if (seconds < 10) {
    Serial.print("0");
  }
  Serial.println(seconds);
}
```

Now for the Loop

At this time we will have a look at the void loop() block of code which is repeated continuously. From here we can control the flow of the sketch so that each of the mode can be accessed when needed and also have the indicator LEDs display properly. The details for each mode will be handled within their own void functions as subroutines. Using this approach limits the size thereby increasing the ease to which we can view and understand the main loop. The loop itself starts on rung 172.

Rungs 173 to 178 are used to create a 1 second pulse that will be used by various modes in different ways. Rung 173 set the value of timeNow to the millisecond time located in the internal timer of the Uno. On rung 174 we check to see if this timeNow value is greater than a value called timePulse, which will be 0 the first time through the loop. If the check on rung 174 is true then the code on rungs 175 to 177 is completed. First on rung 175 we set a new value for timePulse which is the sum of timeNow and timeGoal, this has a value of 1000 which is equal to 1000 milliseconds or 1 second. Rung 176 sets a variable called pulseTrue to a true state and can be used by the subroutines. On rung 177 we set a variable called setColon to either true or false which was the opposite state of the variable on the previous loop. This allows us to make the colon on the 7-Segment Display blink with a 1 second pulse if we need to in any of the modes.

Rung 180 is used to copy the RTC real time for use in modes 1 or 2 as needed.

Rungs 181 to 183 are used to read the value of the potentiometer and set the brightness of the 7-Segment Display. Rung 184 is used if additional display as being used.

```
171 // ----------------------------- Main Loop
172 void loop () {
173   timeNow = millis();             // One Second Timer Pulse
174   if (timeNow > timePulse) {
175     timePulse = timeNow + timeGoal;
176     pulseTrue = true;
177     setColon = !setColon;
178   }
179 
180   DateTime now = rtc.now();       // Read Real-Time from RTC
181   potValue = analogRead(potPin);  // Set brightness of displays
182   potValue = map(potValue, 0, 1023, 0, 15);
183   matrix.setBrightness(potValue);
184   // matrix1.setBrightness(potValue); // Used for second display
185 
```

In this next block of code, we read if the mode button is being pressed which is on rung 186 by looking at mode pushbutton input. If the button has been pressed has been pressed the mode pushbutton input will be forced to 0 volts, grounded, returning a false value to the sketch. Remember that we set all of the pushbutton values to true, 5 volts, by using the INPUT_PULLUP command in the void setup(). Rung 187 is used to test this true/false condition but looking for the false state using the !modeState variable. If the input is false we increase the mode value by 1 on rung 188. Then on rung 189 we check to see if the mode value is greater than 5, if it is then we set it back to a value of 1. This limits the valid range for mode from 1 to 5 as these are the number of modes we have.

On rungs 192 and 193 we send a message to the Serial Monitor so that we can see what the current mode is in the sketch. Finally, on Rung 194 we create a ½ second display so that we have time to stop pressing the mode pushbutton and prevent us getting multiple input signals from it. After all we should just advance 1 mode at a time.

```
186  modeState = digitalRead(modePin); // Read Mode PB state
187  if (!modeState) {                 // Toggle through Mode states
188    modeValue = modeValue + 1;
189    if (modeValue > 5) {
190      modeValue = 1;
191    }
192    Serial.print("Mode Value: ");
193    Serial.println(modeValue);
194    delay(500);
195  }
196
```

The next larger block of code on rungs 197 to 230 is used to control what functions will be completed based on the current mode value, 1 to 5. This starts on rung 197 with an instruction that I introduced in the previous Part 4 article. This instruction is the switch(). Here we use the modeValue variable from the code on the rungs above and then we can execute the code inside of the switch() based on the case value. The case value is the value that is located inside of the brackets on the switch().

Each case is using similar code and this can be broken into the following and I will use case 1 as the example. Rung 198 states which case it is looking for. For each case I have added a comment to make it easier to identify each case. The next 3 rungs, 199 to 201 set the LEDs to on or off depended on what we want for each case. Green is used for the Real-Time Clock. Yellow is used when setting can be changed. Blue is used for the Fast-Time Clock. The next piece of code, 3 rungs this time and only 1 rung for the other cases, is used to call or jump to the subroutine needed for this mode. At the end of each case, rung 205 for this case, is a break command. This break command allows to jump free of the switch() without looking at the rest of the cases.

This code repeats for each of the modes which are;
Mode 1: Real-Time Clock Display
Mode 2: Set Real-Time Clock
Mode 3: Fast-Time Clock Display and Run/Pause Time
Mode 4: Set Fast-Time Starting Time
Mode 5: Set Fast-Time Ratio

```
switch(modeValue) {
  case 1:     // Real time clock running mode
    digitalWrite(greenPin, HIGH); // Set LEDs to Green ON
    digitalWrite(bluePin, LOW);
    digitalWrite(yellowPin, LOW);
    if (timePulse) {
      realTime();
    }
    break;
  case 2:     // Real time clock adjust time mode
    digitalWrite(greenPin, HIGH); // Set LEDs to Green & Yellow ON
    digitalWrite(bluePin, LOW);
    digitalWrite(yellowPin, HIGH);
    setRealTime();
    break;
  case 3:     // Fast clock running mode
    digitalWrite(greenPin, LOW);  // Set LEDs to Blue ON
    digitalWrite(bluePin, HIGH);
    digitalWrite(yellowPin, LOW);
    fastClock();
    break;
  case 4:     // Fast clock adjust start time mode
    digitalWrite(greenPin, LOW);  // Set LEDs to Blue & Yellow ON
    digitalWrite(bluePin, HIGH);
    digitalWrite(yellowPin, HIGH);
    setFastClock();
    break;
  case 5:     // Fast time ratio setting mode
    digitalWrite(greenPin, LOW);  // Set LEDs to Blue & Yellow ON
    digitalWrite(bluePin, HIGH);
    digitalWrite(yellowPin, HIGH);
    setFastRatio();
    break;
}
```

Most of the remaining code in the void loop() section is used to reset many of the variables that are used in various mode subroutines. Since it is possible to switch modes at almost any time we need to be able to reset variables that may cause us problems when we re-enter that mode.

This first of these resets is located on rungs 231 to 238. This code is used to reset the variables used in mode 2 when we are setting the time of the Real-Time clock.

```
231 if (modeValue != 2) {
232   setRealTimeRead = false;
233   setYear = false;
234   setMonth = false;
235   setDay = false;
236   setHour = false;
237   setMinute = false;
238 }
```

For the block of code on rungs 239 to 241 only 1 variable needs to be reset and that is to pause the Fast-Time clock when you leave mode 3.

```
239 if (modeValue != 3) { // Set Fast-Time to Pause Mode if not in mode 3
240   ftRunning = false;
241 }
```

This next block of code on rungs 242 to 246 is checking for 2 conditions to be true. If both are true, the Fast-Time is paused and we are in mode 3, then the display will be blinking on and off. This will let our operators that the Fast-Time Clock is currently paused and they should not be operating their trains. If either condition or both conditions are false the display will be on steady and not flashing.

```
242   if ((!ftRunning) && (modeValue == 3)) { // Flash display in mode 3 if paused
243     matrix.blinkRate(2);
244   } else {
245     matrix.blinkRate(0);
246   }
```

This block of code on rungs 247 to 253 are used any time we are not in mode 4 so that we can reset all of the variables required for setting the Fast-Time Clock time so that these functions will work properly when we re-enter mode 4.

```
247   if (modeValue != 4) { // Set Fast-Time settings false if not in mode 4
248     setFastTimeRead = false;
249     setminutesR = false;
250     setminutesL = false;
251     sethoursR = false;
252     sethoursL = false;
253   }
```

This final line of code in the void loop() section is used to reset the pulseTrue variable that we set back on rung 176.

```
254   pulseTrue = false;    // Reset 1 second pulse
255 }
256
```

This completes this section of coding and the only part remaining is each of the subroutines.

Mode 1 Subroutine

The mode 1 subroutine is used to display the Real-Time Clock on the display. This section of code is not that complex since none of the buttons, except the mode button, can do anything while in this mode. The task is simpler, just display the time.

To display the Real-Time, we need to breakdown the minutes and hours to individual digits, one for each of the 7-Segment sections on the display. We will start with start with the hours. The left most digit only has 3 possible numbers that can be displayed, either 0, 1 or 2. On rung 258 we start with the name of the subroutine, void realTime(). Next we get the current date and time from the RTC module and store this in the default variable now. On rung 260 we check the value of the hours component by looking at now.hour(). If the now.hour() is less than 10 we place a 0 in the left most digit and the right digit of the variable in the next digit, second from the left. If the value is 10 or more and we move on to rung 265 through the use of the else instruction. Again, we move on if the value is 20 or more. This will complete the hours display as the maximum hours value would be 23. If it was 24 the number would reset to 0.

Now on rung 274 we tell the colon in the middle of the display to remain on steady and not to flash.

```
257 //----------------------------------- Used to display the Real-Time
258 void realTime() {
259     DateTime now = rtc.now();
260     if (now.hour() < 10)
261       {
262         matrix.writeDigitNum(0, 0);
263         matrix.writeDigitNum(1, now.hour());
264       }else{
265         if (now.hour() < 20)
266         {
267           matrix.writeDigitNum(0, 1);
268           matrix.writeDigitNum(1, (now.hour() - 10));
269         }else{
270           matrix.writeDigitNum(0, 2);
271           matrix.writeDigitNum(1, (now.hour() - 20));
272         }
273       }
274     matrix.drawColon(true);
```

Starting on rung 275 and continuing to rung 309 we display the minute using code similar to that of the hours displayed. Rung 310 sends all the information to the display.

```
275     if (now.minute() < 10)
276     {
277       matrix.writeDigitNum(3, 0);
278       matrix.writeDigitNum(4, now.minute());
279     }else{
280       if (now.minute() < 20)
281       {
282         matrix.writeDigitNum(3, 1);
283         matrix.writeDigitNum(4, (now.minute() - 10));
284       }else{
285         if (now.minute() < 30)
286         {
287           matrix.writeDigitNum(3, 2);
288           matrix.writeDigitNum(4, (now.minute() - 20));
289         }else{
290           if (now.minute() < 40)
291           {
292             matrix.writeDigitNum(3, 3);
293             matrix.writeDigitNum(4, (now.minute() - 30));
294           }else{
295             if (now.minute() < 50)
296             {
297               matrix.writeDigitNum(3, 4);
298               matrix.writeDigitNum(4, (now.minute() - 40));
299             }else{
300               if (now.minute() < 60)
301               {
302                 matrix.writeDigitNum(3, 5);
303                 matrix.writeDigitNum(4, (now.minute() - 50));
304               }
305             }
306           }
307         }
308       }
309     }
310     matrix.writeDisplay();
311 }
```

Mode 2 Subroutine

The mode 2 subroutine is used to set the date and time for the Real-Time. This is one of the longest sections of code since there are many variables to set in the RTC module. The one good part about using this setting is that each variables are set separately and you should not be doing this very often.

We start on rung 313 by naming the subroutine, setRealTime(). In rungs 314 to 323 we check to see that we have not yet read in the variables from the RTC module and have stored them in some temporary variables. Once we read in the date and time on rung 315 we break down each of the date and time variables to out temporary variable on rungs 316 to 320. Next, we set a Boolean value, setYear, on rung 321 to true as it will be the first of the date and time variables we can change. We also set the Boolean value setRealTimeRead on rung 322 to true so that we only do this reading and setting of the variables once while in mode 2.

```
312 //--------------------------------------- Used to Set the Real-Time
313 void setRealTime() {
314   if (!setRealTimeRead) {            // Read the current Real-Time
315     DateTime now = rtc.now();
316     wasYear = now.year();
317     wasMonth = now.month();
318     wasDay = now.day();
319     wasHour = now.hour();
320     wasMinute = now.minute();
321     setYear = true;
322     setRealTimeRead = true;
323   }
```

In the next block of code we read the states of the pushbuttons excluding the mode button. This takes place on rungs 324 to 326. The run and pause buttons are used as the up and down buttons when setting any variables. The enter button is used to accept the new value for the variable currently being adjusted and moves us to the next step. These steps are shown in the simplified flow chart back on the first page.

Our first variable to adjust will be the year which is done on rungs 327 to 343. On rung 327 we check and make sure that we have read in the current values for each variable in the RTC date and time by checking the state of setRealTimeRead as being true. This was set on rung 322 of the block of code above. We also check to make sure we are currently in the year variable by check the state of setYear as being true. This was set on rung 321 above.

The next 2 if(), on rungs 328 and 332 check to see if the up or down button is currently pressed. If one of these buttons have been pressed the year value will increase or decrease. There are no upper of lower limits for the value but I don’t suspect you are going to push the button too many times.

Once the desired year value is displayed it is time to press the enter button. When this occurs next the block of code for adjusting the month variables is set to true, setMonth on rung 337, the year block of code is set to false, setYear on 338. Lastly we a slight pause of 300 milliseconds to give you time to let go of the enter button.

Rungs 341 and 342 are used to display the current value for the year on the 7-Segment Display. This is updated constantly while setting the year variable.

```
324 runState = digitalRead(runPin);     // Read state of UP button
325 pauseState = digitalRead(pausePin); // Read state of DOWN button
326 enterState = digitalRead(enterPin); // Read state of ENTER button
327 if ((setRealTimeRead) && (setYear)) {   // Set Year
328   if (!runState) {
329     wasYear = wasYear + 1;
330     delay(150);
331   }
332   if (!pauseState) {
333     wasYear = wasYear - 1;
334     delay(150);
335   }
336   if (!enterState) {
337     setMonth = true;
338     setYear = false;
339     delay(300);
340   }
341   matrix.print(wasYear);
342   matrix.writeDisplay();
343 }
```

The block of code on rungs 344 to 365 are used to set the value of the month variable so there really isn't any point going through this in detail as it works the same as setting the value of the year variable. A range of 1 to 12 has been added since there are only 12 months in a year.

```
344 enterState = digitalRead(enterPin); // Read state of ENTER button
345 if ((setRealTimeRead) && (setMonth)) {  // Set Month
346   if (!runState) {
347     if (wasMonth < 12) {
348       wasMonth = wasMonth + 1;
349       delay(150);
350     }
351   }
352   if (!pauseState) {
353     if (wasMonth > 1) {
354       wasMonth = wasMonth - 1;
355       delay(150);
356     }
357   }
358   if (!enterState) {
359     setDay = true;
360     setMonth = false;
361     delay(300);
362   }
363   matrix.print(wasMonth);
364   matrix.writeDisplay();
365 }
```

In the next block of code on rungs 366 to 397 we adjust the day value. The only difference with this code from the previous is that I've set maximum limits based on the possible number of days in each month.

```
366 enterState = digitalRead(enterPin); // Read state of ENTER button
367 if ((setRealTimeRead) && (setDay)) {    // Set Day
368   if (!runState) {
369     if (wasMonth == 2) {
370       if (wasDay < 29) {
371         wasDay = wasDay + 1;
372       }
373     } else if ((wasMonth == 4) || (wasMonth == 6) || (wasMonth == 9) || (wasMonth == 11)) {
374       if (wasDay < 30) {
375         wasDay = wasDay + 1;
376       }
377     } else {
378       if (wasDay < 31) {
379         wasDay = wasDay + 1;
380       }
381     }
382     delay(150);
383   }
384   if (!pauseState) {
385     if (wasDay > 1) {
386       wasDay = wasDay - 1;
387       delay(150);
388     }
389   }
390   if (!enterState) {
391     setHour = true;
392     setDay = false;
393     delay(300);
394   }
395   matrix.print(wasDay);
396   matrix.writeDisplay();
397 }
```

Now it is time to set the value of the hours variables using the code in rungs 389 to 419. Again, limits are set up to limit the range of the hours.

```
398 enterState = digitalRead(enterPin); // Read state of ENTER button
399 if ((setRealTimeRead) && (setHour)) {  // Set Hour
400   if (!runState) {
401     if (wasHour < 24) {
402       wasHour = wasHour + 1;
403       delay(150);
404     }
405   }
406   if (!pauseState) {
407     if (wasHour > 0) {
408       wasHour = wasHour - 1;
409       delay(150);
410     }
411   }
412   if (!enterState) {
413     setMinute = true;
414     setHour = false;
415     delay(300);
416   }
417   matrix.print(wasHour);
418   matrix.writeDisplay();
419 }
```

There is only one last variable to adjust and that is for the minutes. The code is in rungs 420 to 441. Limits have also been included for the minutes.

```
  enterState = digitalRead(enterPin); // Read state of ENTER button
  if ((setRealTimeRead) && (setMinute)) {  // Set Hour
    if (!runState) {
      if (wasMinute < 59) {
        wasMinute = wasMinute + 1;
        delay(150);
      }
    }
    if (!pauseState) {
      if (wasMinute > 0) {
        wasMinute = wasMinute - 1;
        delay(150);
      }
    }
    if (!enterState) {
      setRealTimeWrite = true;
      setMinute = false;
      delay(300);
    }
    matrix.print(wasMinute);
    matrix.writeDisplay();
  }
```

Now that all of the values for the Real-Time have been adjusted the only thing left to do in mode 2 is to save these values in the Real-Time Clock module. The code in the rungs 442 to 451 handles this need. The code on rung 443 access the RTC module and is the instruction that accomplishes the saving function. Rung 444 includes a slight delay to allow the save enough time before continuing. Rung 445 set the display into a full display blinking mode. The cause all segments that are used to display the time to flash between on and off. This lasts for 1 second as set by the code on rung 447. The blink function of the display is turned off by setting the blinkRate to 0 on rung 448. Rung 449 sets the state of setRealTimeWrite to false so the information is only saved once on the RTC module.

```
442   if ((setRealTimeRead) && (setRealTimeWrite)) {
443     rtc.adjust(DateTime(wasYear, wasMonth, wasDay, wasHour, wasMinute, 0));
444     delay(100);
445     matrix.blinkRate(1);
446     realTime();
447     delay(1000);
448     matrix.blinkRate(0);
449     setRealTimeWrite = false;
450   }
451 }
452
```

Well that’s it for the code required for mode 2, setting the Real-Time Clock. This was long but well worth it as the time may not always be correct. Depending on where you live you may need to take in account changes for Daylight Savings Time. Also, if the battery dies the time may not be working when the unit is off-line. Be sure to have the battery installed in the RTC module even if the battery is dead as it needs to know that one has been installed in order to operate.

Mode 3 Subroutine

The mode 3 subroutine is used to run and display the Fast-Time. The options this time are restricted to actively running the Fast-Time or pausing the Fast-Time. No other adjustments can be made in this mode.

This subroutine starts with a comment on rung 453 to let us know where it starts. Rung 454 is used to define the name of the subroutine as void fastClock(). Rungs 455 and 456 are used to check the state of the pushbuttons for the run and pause functions. At this point both pushbuttons should be returning a true condition since neither is being pressed and we set the input pins to pinMode state of INPUT_PULLUP back on rungs 86 and 87 in the void setup().

Rungs 457 to 459 will set the value of ftRunning to true if the run buttons is pressed. Rungs 460 to 462 will set the value of ftRunning to false if the pause button is pressed. The value of ftRunning is maintained in the last state chosen if not buttons are being pressed. Also, anytime we are not in mode 3 the value of ftRunning will be false as this was set back on rungs 239 to 241 in the void loop().

```
453 //------------------------------------- Used to Dsiaplay Fast-Clock
454 void fastClock() {
455   runState = digitalRead(runPin);     // Read state of RUN button
456   pauseState = digitalRead(pausePin); // Read state of PAUSE button
457   if (!runState) {
458     ftRunning = true;
459   }
460   if (!pauseState) {
461     ftRunning = false;
462   }
463
```

The next block of code starting on rung 464 is used to calculate each digit of the Fast-Time when the ftRunning value is true and when we see the 1 second pulse that was created on rungs 173 to 178 in the void loop().

The calculation begins on rung 465 by adding the ratio value, rVal, to the seconds value to come up with a new seconds total. If the seconds value was 17 and we add the ratio value, let's say the ratio is set to 8:1, the new seconds total would be 25. On the following pulses we would again add 8 for each pulse. The resulting totals would be 33, then 41, 49, 57, 65 and so on. We need to do something at this point as the total is now over 59 and as we know 60 seconds equal 1 minute. So, at this point we increase the value of the minutes by 1 and subtract 60 from the seconds. Doing this the new seconds value would be 5 and no longer would it be 65.

We continue doing similar calculations for each of the 4 digits to be displayed. We first look at the right minutes digit value remembering that valid values for this digit are 0 to 9. Next, we do the calculation for the left minutes digit which has a valid range of 0 to 5. The hours right digit is next with a range from 0 to 9 if the left hours digit is a 0 or 1 but a lesser range of 0 to 3 when the left hours digit is 3. Finally we do the left hours digit which has a value of 0 to 2. As we get close to midnight the time would be 23:59 so when enough seconds have been accumulated to add 1 more minute the display would roll over to 00:00.

```
464 if ((pulseTrue) && (ftRunning)) {
465   seconds = seconds + rVal;
466   if (seconds > 59) {
467     minutesR = minutesR + 1;
468     seconds = seconds - 60;
469   }
470   if (minutesR > 9) {
471     minutesL = minutesL + 1;
472     minutesR = 0;
473   }
474   if (minutesL > 5) {
475     hoursR = hoursR + 1;
476     minutesL = 0;
477   }
478   if ((hoursR > 9) || ((hoursL == 2) && (hoursR > 3))) {
479     hoursL = hoursL + 1;
480     hoursR = 0;
481   }
482   if (hoursL > 2) {
483     hoursL = 0;
484   }
```

Now that we have calculated each digit of the display and the seconds we need to save this value to the eeprom memory located on the Uno microprocessor. We complete the saving of these values on rungs 485 to 489.

If you have your computer connected to the Uno and have uploaded the sketch to the Uno you can use the Serial Monitor to watch the Fast-Time operate. This can be helpful if you are trying to understand the operation of the Fast-Time or if you are having any problems in running it.

As a reminder both the eeprom and print instructions are located inside of the if statement so this is only done once per second when the pulse is true.

```
485   EEPROM.write(fthLAddress, hoursL);
486   EEPROM.write(fthRAddress, hoursR);
487   EEPROM.write(ftmLAddress, minutesL);
488   EEPROM.write(ftmRAddress, minutesR);
489   EEPROM.write(ftsecondsAddress, seconds);
490
491   Serial.print("Fast-Time = ");
492   Serial.print(hoursL);
493   Serial.print(hoursR);
494   Serial.print(":");
495   Serial.print(minutesL);
496   Serial.print(minutesR);
497   Serial.print(":");
498   if (seconds < 10) {
499     Serial.print("0");
500   }
501   Serial.println(seconds);
502 }
```

The final block of code in this subroutine is used to send the time to the 7-Segment Display. The display is updated during each time we loop through the sketch while we are in mode 3.

```
  matrix.writeDigitNum(0, hoursL);
  matrix.writeDigitNum(1, hoursR);
  matrix.drawColon(true);
  matrix.writeDigitNum(3, minutesL);
  matrix.writeDigitNum(4, minutesR);
  matrix.writeDisplay();
}

```

Mode 4 Subroutine

In the mode 4 subroutine we will be able to set the current Fast-Time Clock setting. This means that we can adjust the time that will be used begin running your Fast-Time. If you only run daytime operating sessions you could set the start time for the session at 06:00 or 07:30 as examples. The beginning time for your session is your choice.

Similar to the other subroutines I start with a comment to identify the beginning of the subroutine on rung 511 and then name the subroutine void seFastClock() on rung 512. From here on rungs 513 to 522 we read in the currently saved Fast-Time from the eeprom memory. At this point we set a couple of variables to true for use in the next few blocks of code and we display the current value of sethoursL on the Serial Monitor if it is being used. All of this is similar to what we did in mode 2 for setting the Real-Time.

```
511 //-------------------------------------- Used to Set the Fast-Clock
512 void setFastClock() {
513   if (!setFastTimeRead) {            // Read the current Fast-Time
514     hoursL = EEPROM.read(fthLAddress);
515     hoursR = EEPROM.read(fthRAddress);
516     minutesL = EEPROM.read(ftmLAddress);
517     minutesR = EEPROM.read(ftmRAddress);
518     sethoursL = true;
519     setFastTimeRead = true;
520     Serial.print("sethoursL = ");
521     Serial.println(sethoursL);
522   }
```

Now it is time to read in the states of each pushbutton, rungs 523 to 525, so that they can be used while in this mode.

```
523   runState = digitalRead(runPin);     // Read state of UP button
524   pauseState = digitalRead(pausePin); // Read state of DOWN button
525   enterState = digitalRead(enterPin); // Read state of ENTER button
526
```

On rungs 527 to 548 we will set the value of the left hours digit. We need to make sure that the value remains within a valid range which is very limited for this digit, only from 0 to 1. Next we save the value to the eeprom memory and display the sethoursR value to the Serial Monitor. We also set the value of sethoursL to false and sethoursR to true to move to the next block of code for adjusting the right hours digit. The steps we follow for setting the Fast-Time is very similar to that of setting the Real-Time so the code should be easy to follow.

```
if (setFastTimeRead && sethoursL) { // Set hoursL
  if (!runState) {
    if (hoursL < 2) {
      hoursL = hoursL + 1;
      delay(150);
    }
  }
  if (!pauseState) {
    if (hoursL > 0) {
      hoursL = hoursL - 1;
      delay(150);
    }
  }
  if (!enterState) {
    EEPROM.write(fthLAddress, hoursL);  // Save hoursL to eeprom memory
    sethoursR = true;
    sethoursL = false;
    delay(500);
    Serial.print("sethoursR = ");
    Serial.println(sethoursR);
  }
}
```

Now we set the right hours digit then move onto the left minutes digit following the same coding pattern as above.

```
enterState = digitalRead(enterPin); // Read state of ENTER button
if (setFastTimeRead && sethoursR) { // Set hoursR
  if ((!runState) && (hoursL > 1)) {
    if (hoursR < 4) {
      hoursR = hoursR + 1;
      delay(150);
    }
  }
  if ((!runState) && (hoursL < 2)){
    if (hoursR < 9) {
      hoursR = hoursR + 1;
      delay(150);
    }
  }
  if (!pauseState) {
    if (hoursR > 0) {
      hoursR = hoursR - 1;
      delay(150);
    }
  }
  if (!enterState) {
    EEPROM.write(fthRAddress, hoursR);  // Save hoursR to eeprom memory
    setminutesL = true;
    sethoursR = false;
    delay(500);
    Serial.print("setminutesL = ");
    Serial.println(setminutesL);
  }
}
```

Now we set the left minutes digit then move onto the right minutes digit following the same coding pattern as above.

```
enterState = digitalRead(enterPin); // Read state of ENTER button
if (setFastTimeRead && setminutesL) { // Set minutesL
  if (!runState) {
    if (minutesL < 5) {
      minutesL = minutesL + 1;
      delay(150);
    }
  }
  if (!pauseState) {
    if (minutesL > 0) {
      minutesL = minutesL - 1;
      delay(150);
    }
  }
  if (!enterState) {
    EEPROM.write(ftmLAddress, minutesL);  // Save minutesL to eeprom memory
    setminutesR = true;
    setminutesL = false;
    delay(500);
    Serial.print("setminutesR = ");
    Serial.println(setminutesR);
  }
}
```

Now we set the right minutes digit then move onto the left hours digit following the same coding pattern as above. This time it allows us to return to the left hours digit so that we can keep adjusting all of the digits until we are satisfied by the time have entered. Pressing the mode button we take us out of this loop and on to the subroutine for adjusting the Fast-Time ratio.

```
enterState = digitalRead(enterPin); // Read state of ENTER button
if (setFastTimeRead && setminutesR) { // Set minutesR
  if (!runState) {
    if (minutesR < 9) {
      minutesR = minutesR + 1;
      delay(150);
    }
  }
  if (!pauseState) {
    if (minutesR > 0) {
      minutesR = minutesR - 1;
      delay(150);
    }
  }
  if (!enterState) {
    EEPROM.write(ftmRAddress, minutesR);  // Save minutesR to eeprom memory
    sethoursL = true;
    setminutesR = false;
    delay(500);
    Serial.print("sethoursL = ");
    Serial.println(sethoursL);
  }
}
```

In the next block of code on rungs 624 to 646 we control what is being displayed on the 7-Segment display as we work through setting each of the digits. One thing to note is that on the 7-Segment Display a small dot that is located at the bottom right corner of each digit. When the dot is lit up that lets us know which digit we are adjusting. The 3 remaining dots will be off at this time.

```
  if (sethoursL) {
    matrix.writeDigitNum(0, hoursL, true);
  } else {
    matrix.writeDigitNum(0, hoursL);
  }
  if (sethoursR) {
    matrix.writeDigitNum(1, hoursR, true);
  } else {
    matrix.writeDigitNum(1, hoursR);
  }
  matrix.drawColon(setColon);
  if (setminutesL) {
    matrix.writeDigitNum(3, minutesL, true);
  } else {
    matrix.writeDigitNum(3, minutesL);
  }
  if (setminutesR) {
    matrix.writeDigitNum(4, minutesR, true);
  } else {
    matrix.writeDigitNum(4, minutesR);
  }
  matrix.writeDisplay();
}

```

Mode 5 Subroutine

We are just about at the end of the sketch as this is the last subroutine and block of code that is needed to make this project work. This subroutine is much shorter in length than the previous subroutines and it is located on rungs 648 to 686. The purpose of this subroutine is to set the Fast-Time ratio to a value that falls within the range from 1:1 to 12:1.

On rung 648 I have included a comment to make it easier to find the beginning of the subroutine. The subroutine is named, void setFastRatio(), on rung 649. Rungs 650 to 663 are used to display the current ratio on the 7-Segment Display. The value in memory was read from the eeprom back in the void setup() section.

```
648 //------------------------- Used to Set the Ratio of the Fast-Clock
649 void setFastRatio(){
650   if (ratioValue < 10) {                // Display the ratio, xx:01
651     matrix.writeDigitNum(0, 0);
652   }else{
653     matrix.writeDigitNum(0, 1);
654   }
655   if (ratioValue < 10) {
656     matrix.writeDigitNum(1, ratioValue);
657   }else{
658     matrix.writeDigitNum(1, ratioValue - 10);
659   }
660   matrix.drawColon(setColon);
661   matrix.writeDigitNum(3, 0);
662   matrix.writeDigitNum(4, 1);
663   matrix.writeDisplay();
```

In the next block of code, rungs 664 and 665, the state of the pushbuttons for run (up) and pause (down) are checked. The next few rungs, 666 to 671, is used to increase the value of the ratio as high as 12. Rungs 672 to 677 is used to decrease the value of the ratio to a low of 0.

```
664   runState = digitalRead(runPin);     // Read state of UP button
665   pauseState = digitalRead(pausePin); // Read state of DOWN button
666   if (!runState) {                    // Increase ratio value
667     if (ratioValue < 12) {
668       ratioValue = ratioValue + 1;
669       delay(250);
670     }
671   }
672   if (!pauseState) {                  // Decrease ratio value
673     if (ratioValue > 1) {
674       ratioValue = ratioValue - 1;
675       delay(250);
676     }
677   }
```

The final block of code on rungs 678 to 686 are used read the state of the enter button, display the ratio on the Serial monitor and save the value to the eeprom memory.

```
678   enterState = digitalRead(enterPin); // Read state of ENTER button
679   if (!enterState) {                  // Save ratio to eeprom memory
680     rVal = ratioValue;
681     Serial.print("Written rVal = ");
682     Serial.println(rVal);
683     EEPROM.write(rAddress, rVal);
684     delay(300);
685   }
686 }
687
```

Conclusion

This wraps up the code needed to make the 7-Segment Display usable for both Real-Time and Fast-Time display. You can add to the programming by adding code for a second display. To ease the amount of programming you might only include display information needed when in modes 1 or 2, which are used to display times. There really is no need to have a second display functioning when you are setting the Real-Time, Fast-Time or Ratio for the Fast-Time as your operators don't need this information.

Located on the CARM website where you found this expanded article you will also find a video where I show the functions of this project as well as a downloadable file of the completed project as an Arduino file using the ino extension. I will also include other links for images of the project mounted including a second display. I will also add files with updates to code for additional displays.

Well that's if for now, enjoy this and future articles. For the next article in The Canadian I will be building a working (moving spout) water tank.

If you have any other project ideas or questions please contact me. Thanks.

David King, directordavid@caorm.org