NANYANG POLYTECHNIC
EG431D Data Acquisition

Lesson 6

Course: Diploma in Engineering (Electronic and Digital Engineering)

Module: EG431D Data Acquisition

Title: Time Domain Analysis

Objective:

The objective of this lesson is to analyze signals in the time domain by focusing on their key attributes such as amplitude, phase, and duration, and their practical relevance in signal processing applications. It aims to explain the fundamental principles of time-domain analysis, including concepts like periodicity, time shifts, and convolution, to provide a deeper understanding of signal behavior in real-world systems. Additionally, this study demonstrates the application of time-domain analysis techniques, such as calculating mean, variance, and autocorrelation, to characterize signals effectively, enabling the extraction of valuable insights for problem-solving in signal processing.

Learning Objectives:

- Describe the process of analyzing signals in the time domain, focusing on key attributes such as amplitude, phase, and duration, and their relevance in practical signal processing applications.
- Explain the fundamental rules of time domain analysis, including key concepts like periodicity, time shifts, and convolution, and how they are applied to understand the behavior of signals in real-world systems.
- Demonstrate how to apply time domain analysis techniques, such as calculating mean, variance, and autocorrelation, to characterize signals and extract useful information for problem-solving in signal processing.

1. Introduction

Time domain analysis is a fundamental technique in signal processing and system analysis, focusing on how a signal or system response evolves over time. It is particularly useful for examining dynamic systems, where the behavior changes continuously or discretely as a function of time. By analyzing how signals vary with time, engineers and scientists can gain insights into transient behaviors, steady-state conditions, and overall system performance.

In the time domain, signals are represented as functions of time, denoted as $\boldsymbol{x(t)}$ for **continuous signals** or $\boldsymbol{x[n]}$ for **discrete signals**. Continuous signals, such as temperature variations or electrical voltages, vary smoothly over time and are typically encountered in analog systems. Discrete signals, on the other hand, result from sampling continuous signals at specific intervals and are prevalent in digital systems. Time-domain representation provides a direct and intuitive view of how a signal behaves in real-world applications, making it an essential tool for monitoring and analysis.

2. Signal Attributes

Key attributes of signals analyzed in the time domain include **amplitude**, **duration**, and **waveform shape**. Amplitude reflects the signal's magnitude at a given moment, providing a measure of its strength. The duration indicates the length of time over which the signal exists, or an event occurs, while the waveform shape describes the signal's overall appearance, such as sinusoidal, square, or arbitrary patterns. These characteristics allow engineers to assess signal properties, detect anomalies, and evaluate system responses.

Several popular **metrics** are used to analyze signals in the time domain, providing insights into their properties and behavior:

a. **Period**: The time it takes for one complete cycle of a periodic signal.

b. **Duration**: The total time over which the signal exists, or an event occurs. Used to analyze transient phenomena, such as a pulse or spike.

c. **Time Interval**: The elapsed time between two significant events in the signal, such as consecutive peaks or zero crossings. Often used to calculate the period of periodic signals.

d. **Rise and Fall Times**: These metrics measure how quickly a signal transitions between defined levels. **Rise Time**: 10% to 90% of its amplitude. **Fall Time**: 90% to 10% of its amplitude.

e. **Settling Time**: In control systems, this metric indicates how quickly a signal reaches and remains within a specific range after a disturbance.

f. **Transient**: Metrics specific to short-term changes in the signal. **Overshoot**: The amount by which the signal exceeds its final steady-state value during a transient response. **Undershoot**: The amount by which the signal dips below its final steady-state value. **Steady-State Error**: The difference between the actual and desired final value.

g. **Zero-Crossing Rate**: The frequency of crossings at the zero level is used to infer periodicity or high-frequency content in the signal.

h. **Phase Information**: The relative position of the signal waveform in time with respect to a reference, particularly important in time-synchronized systems.

i. **Amplitude**: **Peak amplitude** indicates the maximum excursion of the signal from its mean value, providing a measure of signal strength. **Minimum Amplitude** indicates the lowest value of the signal, useful for understanding the range of variation. **Mean Amplitude** indicates the average value of the signal over a specific period, representing the signal's baseline or offset.

j. **Skewness and Symmetry**: **Skewness** measures the asymmetry of the signal waveform around its mean. **Symmetry** helps determine whether the signal's shape is balanced over time.

k. **Signal Envelope**: The upper and lower bounds of the signal, often used to represent amplitude modulation or non-stationary signals.

l. **Signal Energy**: The total energy contained in the signal, calculated as:

$$\boldsymbol{E = \int_{-\infty}^{\infty} x^2(t)dt}$$

Where $\boldsymbol{x(t)}$ is the signal.

This formula is used to determine the **total energy** of a signal, which is particularly relevant for finite-energy signals. It quantifies how much "effort" or "power" the signal carries over its entire existence.

m. **Mean and RMS Values**: The **mean value** represents the average signal level, while the **root mean square** (**RMS**) value quantifies its effective power. For discrete signal ($\boldsymbol{x_i}$), RMS can be estimated as:

$$\boldsymbol{RMS = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}}$$

Where $\boldsymbol{N}$ is the number of samples, and $\boldsymbol{x_i}$ are the individual sample signal values.

n. **Crest Factor:** The ratio of the peak amplitude to the RMS value:

$$\boldsymbol{Crest\ Factor = \frac{Peak\ Amplitude}{RMS}}$$

It indicates the presence of impulsive vibrations, which are often caused by defects like cracks or misalignments in rotating machinery.

o. **Standard Deviation**: A statistical measure that quantifies the amount of variation or dispersion in the signal's amplitude values relative to its mean. It indicates how much the signal fluctuates around its average value, providing insight into the signal's stability or variability. Mathematically, the standard deviation ($\boldsymbol{\sigma}$) is calculated as the square root of the variance, which is the average of the squared differences between each signal value ($\boldsymbol{x_i}$) and the mean ($\boldsymbol{\mu}$):

$$\boldsymbol{\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}}$$

Where $\boldsymbol{N}$ is the number of samples in the signal.

A low standard deviation indicates that the signal values are closely clustered around the mean, while a high standard deviation signifies greater spread or variability. In signal processing, the standard deviation is often used to assess noise levels, signal stability, and fluctuations, playing a critical role in analyzing and characterizing the properties of the signal.

p. **Signal Variance**: A statistical measure that quantifies the spread or dispersion of its amplitude values around the mean. It provides insight into the overall variability of the signal. Mathematically, the variance ($\boldsymbol{\sigma^2}$) is calculated as the average of a squared differences between each signal ($\boldsymbol{x_i}$) and the mean ($\boldsymbol{\mu}$):

$$\boldsymbol{\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

Where $\boldsymbol{N}$ is the number of samples in the signal.

A high variance indicates that the signal values are widely spread out, while a low variance suggests that the values are tightly clustered around the mean. In signal processing, variance is a crucial metric for analyzing the energy or power of fluctuations in a signal, distinguishing between noise and the actual signal content, and characterizing signal stability and consistency.

q. **Kurtosis:** A measure of the "tailedness" or extreme values in a data distribution, indicating how much of the variance is due to outliers. A normal distribution has a kurtosis of 3, with balanced tails. Values above 3 (leptokurtic) suggest heavy tails and more outliers, while values below 3 (platykurtic) indicate lighter tails and fewer outliers. In signal processing and vibration analysis, kurtosis helps identify anomalies, such as sudden impacts or bursts of energy, making it a useful tool for detecting faults or irregularities in systems.

Time domain analysis offers a direct and immediate understanding of how systems and signals behave over time. Its ability to capture transient events, such as spikes, oscillations, or system instabilities, makes it indispensable for applications requiring real-time monitoring and rapid diagnosis. Compared to frequency domain analysis, which focuses on the spectral content of signals, time domain analysis excels in capturing temporal changes and non-periodic phenomena, forming the foundation for advanced techniques like Fourier transforms and predictive modeling.

3. Time Domain Analysis with Arduino

To acquire data for time domain analysis, Arduino boards use ADCs to sample analog signals and convert them into digital values. The *analogRead()* function is commonly employed for this purpose, enabling the capture of data at consistent intervals. Timing functions like *millis()* or *micros()* ensure precise timestamping of data points, maintaining uniform sampling rates. Once captured, the data can be stored in arrays for real-time analysis or transmitted to external systems for further processing.

3.1 Sound Signal Analysis: RMS Measurement

Sound signal analysis is critical in applications such as environmental monitoring, audio signal processing, and machinery health diagnostics. One of the essential metrics for analyzing sound signals is the RMS value, which provides a **measure of the average power of a sound wave** over time. Implementing RMS measurement using an Arduino involves capturing sound data, performing mathematical computations, and presenting the results for analysis.

To begin, we connect a Sound Sensor module (covered in Lesson 2) to an analog pin on the Arduino Board. This sensor captures sound pressure levels and converts them into an analog voltage signal proportional to the intensity of the sound. The Arduino's ADC then digitizes these signals for further processing.

The RMS calculation involves sampling the sound signal over a defined period, squaring each sample, averaging these squared values, and finally taking the square root of the average. This provides a reliable representation of the sound signal's energy, making it less sensitive to instantaneous peaks or troughs.

For discreet audio signals (x_i) sampled at regular intervals, the RMS value is approximated as:

$$\boldsymbol{RMS} = \sqrt{\frac{\mathbf{1}}{\boldsymbol{N}}\sum_{\boldsymbol{i=1}}^{\boldsymbol{N}} \boldsymbol{x_i^2}}$$

Where N is the number of samples.

The calculated RMS value can be displayed on an LCD module, logged to an SD card, or transmitted to a cloud platform for remote monitoring. The Arduino program can also use the RMS value to trigger events, such as sending alerts if the sound level exceeds a predefined threshold, making it useful for noise monitoring or sound-based automation.

For instance, in an environmental monitoring application, the Arduino could continuously compute the RMS value of ambient sound levels and log this data to analyze trends over time. Similarly, in an industrial setting, the RMS value of machinery noise can be monitored to detect anomalies, such as increasing vibration noise, which may indicate wear or impending failure.

3.2 Temperature Analysis: Daily Average, Maximum, Minimum and Rate of Change

Implementing temperature analysis metrics such as daily average, maximum, minimum, and rate of change using an Arduino-based system involves leveraging temperature sensors, data storage, and basic computational logic. The Arduino platform, with its simplicity and versatility, is well-suited for real-time temperature monitoring and analysis in a variety of applications such as environmental monitoring, smart homes, or industrial systems.

To begin, we can interface a Temperature Sensor module (covered in Lesson 2) with the Arduino Board to capture temperature readings. The sensor periodically measures the ambient temperature and sends the data to the Arduino Board for processing. The analog output from the sensor is read through the appropriate Arduino pins and converted into temperature values using predefined calibration formulas or sensor-specific libraries.

For the **daily average temperature**, the Arduino collects temperature readings at regular intervals, such as every minute or hour, and accumulates them in a buffer. At the end of the day, the sum of the readings is divided by the number of samples to compute the average temperature. This metric provides a smooth representation of the daily thermal trend and is useful for identifying overall conditions without being influenced by short-term fluctuations.

For discreet average temperature sampled at regular intervals, the average value is calculated as:

$$\boldsymbol{Daily\ Average\ Temperature} = \frac{\sum_{i=1}^{N} \boldsymbol{T_i}}{\boldsymbol{N}}$$

Where N is the number of samples, and T_i are the individual sampled temperature values.

The **maximum and minimum temperature** metrics are computed by continuously comparing each new reading with the previously stored maximum and minimum values. If the current reading exceeds the maximum or falls below the minimum, the corresponding value is updated. This ensures that the Arduino maintains real-time tracking of the highest and lowest temperatures recorded during the day, enabling applications to respond to extreme thermal conditions promptly.

The **rate of temperature change** is calculated by comparing the current temperature reading with the previous one and determining the difference over time. This metric helps identify sudden changes in temperature, which could indicate system malfunctions, environmental disruptions, or rapid weather changes. By setting thresholds for acceptable rates of change, the system can trigger alarms or corrective actions when abrupt fluctuations occur.

The processed temperature metrics can be displayed using an LCD module, logged to an SD card for historical analysis, or transmitted to a cloud-based dashboard for remote monitoring. Additionally, Arduino can utilize conditional logic to take predefined actions, such as activating cooling systems when maximum temperature thresholds are breached or sending alerts when the rate of change exceeds a safe limit.

3.3 Vibration Analysis

Vibration analysis is a crucial aspect of monitoring the health of machinery, structures, and environments. It helps identify irregularities, such as unbalanced loads, mechanical wear, or structural weaknesses, which can lead to failure if not addressed. An Arduino-based vibration analysis system provides an efficient, cost-effective solution for real-time monitoring, leveraging accelerometers and other sensors to capture and analyze vibration data.

To implement vibration analysis using Arduino, we connect a tri-axial accelerometer (covered in Lesson 2) to the Arduino Board. These accelerometers measure acceleration along the X, Y, and Z axes, providing raw data about the intensity and direction of vibrations. The Arduino Board collects this data using the I2C or SPI interface and processes it to extract meaningful insights.

The first step in vibration analysis is signal acquisition. Arduino reads acceleration data from the sensor at regular intervals, ensuring that the sampling rate adheres to the **Nyquist criterion** for accurate representation of the vibration frequencies. The raw acceleration data may contain noise, which can obscure the actual vibration patterns. To address this, signal preprocessing techniques, such as moving average filtering or low-pass filtering, are applied to smooth the data and remove high-frequency noise.

Once the data is preprocessed, Arduino calculates metrics like the vibration **magnitude**, which is derived from the acceleration components along all three axes. This is done by computing the resultant acceleration using the formula:

$$\boldsymbol{Magnitude} = \sqrt{(\boldsymbol{X^2} + \boldsymbol{Y^2} + \boldsymbol{Z^2})}$$

Where $\boldsymbol{Magnitude}$ provides a single value representing the intensity of the vibration.

Further analysis can include extracting time-domain features such as the **RMS**, **Peak-to-Peak** amplitude, **Crest Factor**, **Kurtosis**, **Skewness**, **Variance** and/or **Standard Deviation** of the vibration signal. These metrics help in quantifying the vibration levels and identifying patterns indicative of potential faults.

Processed data can be visualized using an LCD module or transmitted to a cloud-based dashboard for remote monitoring. Alerts can be programmed to trigger when vibration metrics exceed predefined thresholds, providing timely notifications of potential issues.

3.4 Event Counting

Event counting with an Arduino can be demonstrated using a PIR sensor (covered in Lesson 2). By integrating a PIR sensor with an Arduino, we can create a system that counts **motion events**, making it suitable for applications like monitoring room occupancy, tracking movement in security systems, or gathering activity statistics.

In this implementation, the PIR sensor is connected to one of the Arduino Board's digital input pins. The sensor outputs a HIGH signal whenever motion is detected and remains LOW in the absence of motion. The Arduino program continuously monitors the digital pin for state changes from LOW to HIGH, which signifies the occurrence of a motion event. Each detected event increments a counter variable

stored in the Arduino's memory, maintaining a cumulative record of motion occurrences.

To ensure accuracy, the program includes debouncing logic to prevent false triggering due to sensor noise or rapid state fluctuations. This can be achieved by implementing a short delay or verifying stable states over a brief time window before incrementing the counter. Additionally, the PIR sensor's sensitivity and delay time can often be adjusted via onboard potentiometers to optimize its performance for the specific application.

The event count can be displayed in real-time on an output device such as an LCD module. For advanced applications, the system can also include time-stamped data logging, where each motion event is recorded with its detection time. This data can be transmitted to a cloud-based dashboard for remote monitoring or further analysis. Additionally, the system can integrate features such as setting thresholds to trigger specific actions when a certain number of events are detected, such as sounding an alarm after detecting excessive movement in a security context.

4. Application Examples

3.1 Safe Viewing Distance Enforcement Module for Television

General Requirements

The Safe Viewing Distance Enforcement Module is designed to promote safe viewing practices by monitoring the proximity of users to the television screen and enforcing a minimum safe distance of 2.2 meters. This system integrates an ultrasonic distance sensor (Grove – Ultrasonic Ranger), buzzer, and a timer to detect, alert, and act based on the user's distance from the television. Below are the operational scenarios and their respective behaviors:

1. **Safe Distance Mode (≥ 2.2m)**:

   The module remains dormant when a user is detected at or beyond a safe distance of 2.2 meters. In this mode, no alerts or actions are triggered, allowing the television to function normally.

2. **Reminder Alert Mode (1.5m ≤ Distance < 2.2m)**:

   When the user moves closer to the television, at a distance between 1.5 meters and 2.2 meters, the module activates a reminder alert. The buzzer emits periodic beeps at a 500ms interval to encourage the user to move back to a safer viewing distance.

3. **Warning and Timer Activation Mode (< 1.5m)**:

   If the user comes closer than 1.5 meters to the screen, the module escalates the response:

   - The buzzer switches to a more urgent warning mode, emitting beeps at a 100ms interval.
   - Simultaneously, a 10-seconds timer is activated. If the user does not return to a safe distance (≥ 2.2 meters) within this time, the module powers off the television.

4. **Timer and Buzzer Reset**:

   The buzzer and timer reset immediately when the user moves back to a safe distance of at least 2.2 meters, resuming normal television operation.

5. **Power-Off State**:

   If the timer expires, indicating the user remains too close for 10 seconds, the module powers off the television. In this state:

   - The buzzer is disabled.
   - The television remains powered off until the user moves to a safe distance of at least 2.2 meters, ensuring adherence to safe viewing practices before the television can be powered on again.

This enforcement module enhances safety by integrating real-time distance monitoring with proactive alerts and actions. It uses an ultrasonic distance sensor for precise measurement, an MCU to process data and control outputs, and a buzzer to provide auditory feedback. The timer ensures sufficient warning is given before enforcing the power-off action, promoting healthy viewing habits.

Logic implementation using Finite State Machine

The operational logic of the Safe Viewing Distance Enforcement Module is structured using a Finite State Machine (FSM), as depicted in Figure 3A. This FSM clearly defines the states, transitions, and events that govern the module's behavior. Central to the FSM's operation is the concept of time, which plays a critical role in managing various internal functions such as proximity detection, audible alert generation, and timed actions. By dividing the system's logic into discrete states, the FSM ensures clear, predictable, and efficient control of the module's tasks.

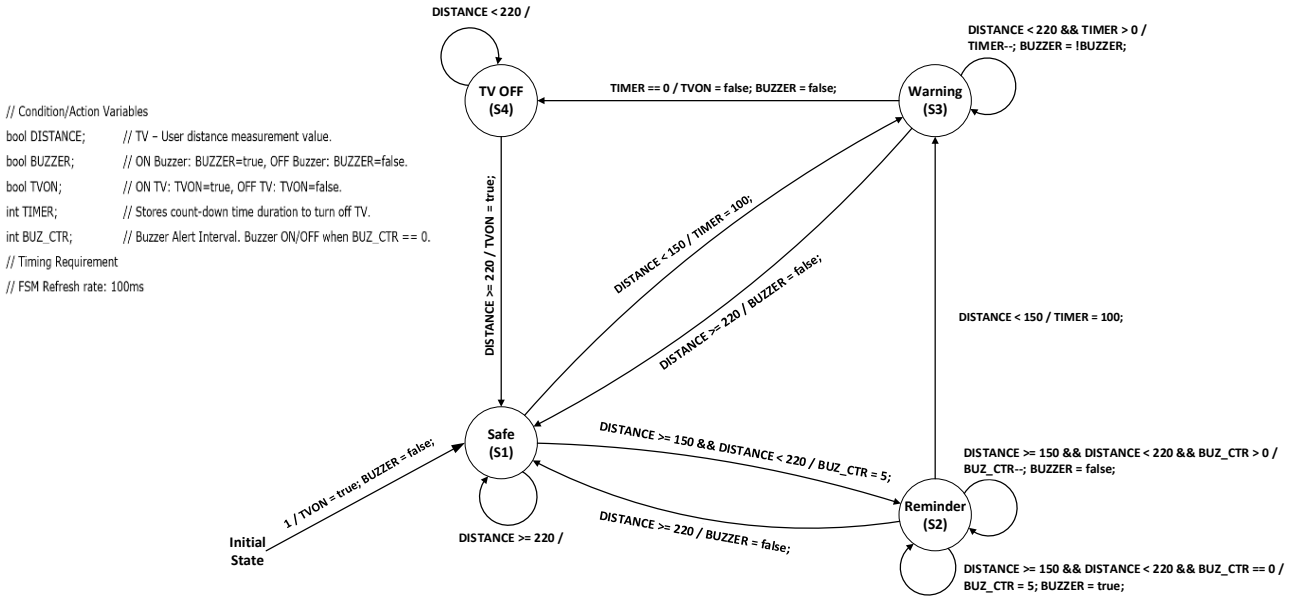

Figure 3A – Finite State Machine of Safe Viewing Distance Enforcement Module

The FSM consists of several states that correspond to different system behaviors. In the **Safe State**, the module remains inactive when the user is at a safe distance of at least 2.2 meters, conserving energy and resources. When the user moves closer,

within the range of 1.5 meters to 2.2 meters, the system transitions to the **Reminder State**, emitting periodic buzzer alerts at 500ms intervals. If the user moves even closer, to a distance of less than 1.5 meters, the module enters the **Warning State**, where it emits higher-frequency buzzer alerts at 100ms intervals and activates a 30-second countdown timer. If the user fails to return to a safe distance within this timeframe, the system transitions to the **TV-OFF State**, turning off the television and requiring the user to move to a safe distance to allowing the television to be turned on again.

The transitions between these states are governed by real-time ultrasonic distance measurements and the passage of time. For example, the module continuously monitors the user's proximity to determine whether to move from the Safe State to the Reminder or Warning States. Time is also critical in the **Power-Off Timer**, which ensures the user has sufficient warning before the television is turned off.

The module's functionality relies on various aspects of **Time Domain Analysis**, and signal pre-processing which ensures accurate and timely responses. Real-time sampling of distance data with pre-processing enables continuous and precise monitoring of the user's position. Audible alert management involves generating buzzer signals at predefined intervals (500ms or 100ms), providing clear feedback on the user's proximity. Additionally, the power-off timer is carefully calibrated to enforce safe viewing distances while minimizing unnecessary interruptions.

Static variables play a key role in managing the system's operations. These variables store critical information such as distance measurements from the sensor, current timer values for the power-off sequence, and state indicators to track the FSM's current position. This structured approach to data management ensures seamless integration between the sensor inputs, state transitions, and system outputs.

3.2 Core Body Temperature Monitor

General Requirements

The Core Body Temperature Monitor is a non-invasive MCU-based temperature monitoring system. The system is designed to continuously monitor core body temperature (CBT) for fever detection in critically ill patients. Utilizing a non-contact AMG8833 infrared thermopile sensors array (TSA) module positioned at 10cm above the patient's forehead with the temporal artery within the sensor's Field-of-View (FoV), this system offers a reliable and efficient means of temperature measurement. The measured target temperature is calculated by using the following formula:

$$T_{Target} = \left[\frac{T_{apparent}^4 - (1-\varepsilon)T_{ambient}^4}{\varepsilon}\right]^{\frac{1}{4}}$$

Where:

- T_{Target} is the calculated (estimated) target object temperature in Kelvin, K.
- $T_{apparent}$ is the measured target object temperature in Kelvin, K.
- $T_{ambient}$ is the measured ambient temperature in Kelvin, K.
- ε is the emissivity of the target object's surface (0.98 for human skin).

The monitoring system incorporates an LCD with RGB backlight to display temperature values and ranges visually and a buzzer to provide audible alerts for

varying levels of fever severity. Operating within a valid temperature range of 36.5 ºC to 41 ºC, the system categorizes body temperature into four states: **Normal Body Temperature**, **Fever**, **Moderate Fever**, and **High Fever**.

The existing **AMG8833 library** lacked support for **ambient temperature readouts**, making it insufficient for this application requiring both environmental and detailed thermal data. The **Adafruit AMG88xx library** addresses this limitation by adding ambient temperature readout functionality alongside the standard **8x8 sensor array data**, enhancing the module's versatility for thermal imaging tasks. This improved library can be downloaded from Adafruit AMG88xx GitHub repository:

https://github.com/adafruit/Adafruit_AMG88xx/tree/master

Logic implementation using Finite State Machine

The system's functionality is governed by an FSM (Figure 3B), ensuring dynamic responses to temperature thresholds by controlling LCD messages, backlight colors, and buzzer operations. When the body temperature (TB) is below 37.5 ºC, the FSM sets the LCD backlight color to green, indicating **Normal Body Temperature**, and silences the buzzer.

For temperatures between 37.5 ºC and 38.3 ºC, the FSM transitions to a **Fever state**, changing the backlight to blue and activating the buzzer at 400ms intervals. In the **Moderate Fever state**, for temperatures ranging from 38.3 ºC to 39 ºC, the LCD backlight turns yellow, and the buzzer beeps at 100ms intervals.

For temperatures of 39 ºC or higher, the FSM enters a critical **High Fever state**, displaying a red backlight and the buzzer emits a continuous warning sound. The LCD screen continuously displays precise temperature readings for caregiver reference. This monitoring system can incorporate an optional Cloud Dashboard where the patient can be remotely monitored by the caregiver.

Pre-processing of temperature data is essential to enhance measurement reliability, minimize noise, and ensure accurate categorization. Time Domain Analysis techniques are employed to identify trends and provide actionable insights into temperature variations over time. This comprehensive approach ensures timely detection of fever severity, offering an effective monitoring solution for critically ill patients while delivering clear, real-time feedback through visual and auditory indicators.

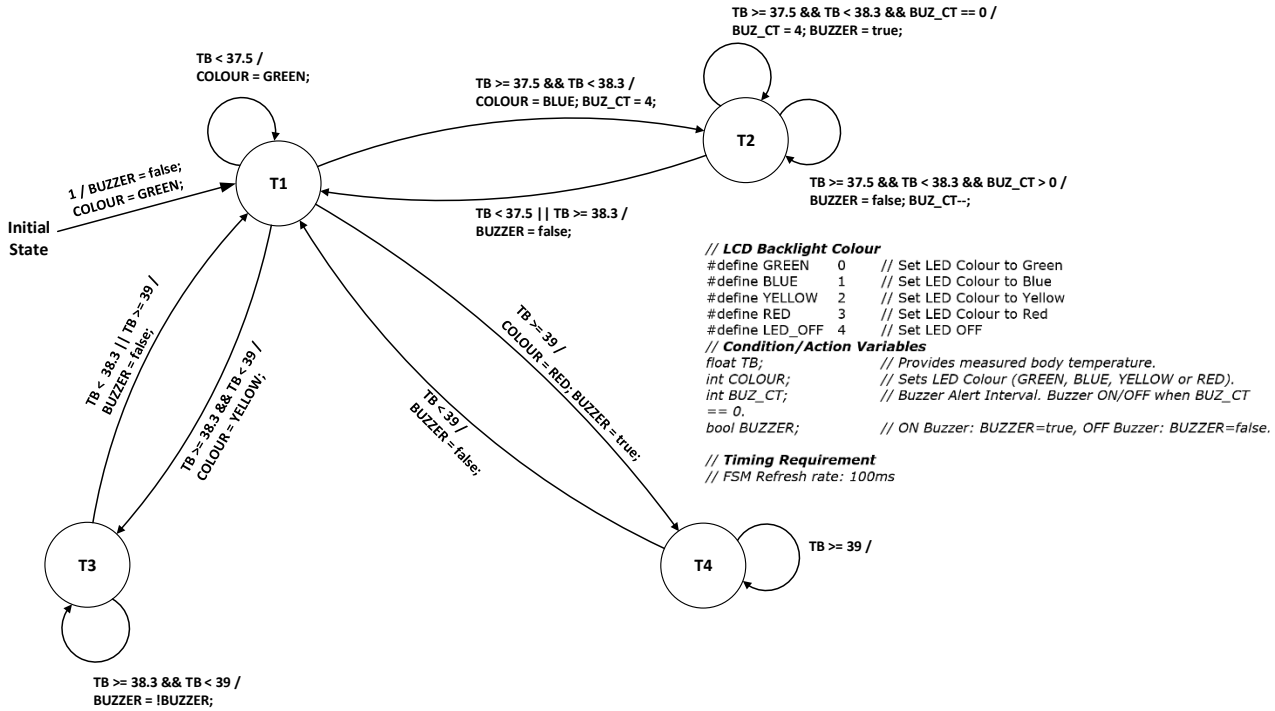

Figure 3B – Finite State Machine of Core Body Temperature Monitor

- The End -