**. Analog-to-Digital Conversion (ADC)**

* **Purpose**: Read voltage and current values from sensors.
* **Implementation**:
  + The ADC on Arduino can be accessed by setting specific bits in the ADCSRA, ADMUX, and other related registers.
  + **Steps**:
    1. Set the reference voltage by configuring ADMUX. For example, to use the 5V reference, set REFS0 to 1.
    2. Select the input channel by setting the lower bits of ADMUX.
    3. Start the conversion by setting the ADSC bit in the ADCSRA register.
    4. Wait for the conversion to complete (check the ADIF bit).
    5. Read the result from the ADC data registers (ADCL and ADCH)

**GPIO Control (LEDs and Buttons)**

* **Purpose**: Control LED indicators and read button states.
* **Implementation**:
  + Use DDRx (Data Direction Register) to set the direction of pins (input/output).
  + Use PORTx to set or clear output pins (LEDs).
  + Use PINx to read the state of input pins (buttons).

**Interrupts**

* **Purpose**: Handle events such as button presses and power threshold alerts.
* **Implementation**:
  + Use EIMSK, EICRA, and EIFR registers to configure and enable external interrupts.
  + For example, an interrupt on pin PD2 can be set up with INT0.

**Counters and Timers**

* **Purpose**: Handle time intervals for sampling data, EEPROM writing, and debouncing.
* **Implementation**:
  + Set up Timer1 or Timer2 using TCCR1A, TCCR1B, and other timer-related registers.
  + Use the OCR1A register to set the compare match value, enabling an interrupt at specified intervals.

**EEPROM**

* **Purpose**: Store cumulative power data to preserve it across power cycles.
* **Implementation**:
  + Access EEPROM directly by writing to EEAR (address register), EEDR (data register), and controlling bits in EECR.

**Serial Communication**

* **Purpose**: Transmit data between zone Arduinos and the central Arduino.
* **Implementation**:
  + Set baud rate by writing to UBRR0H and UBRR0L.
  + Enable transmitter and receiver using UCSR0B, and send/receive data via UDR0.

**Switch Debouncing**

To prevent noise in button presses from causing multiple readings, you’ll need to implement debouncing:

1. **Interrupt-Based Debouncing**:
   * Use a short delay or counter to ignore subsequent button states after the first press.

**Timer-Based Debouncing**:

* Set a timer to enable the button interrupt only after a stable reading.

### Direct Register Access for Overall Optimization

Using direct register access bypasses Arduino’s high-level library functions (pinMode, digitalWrite, etc.), achieving finer control over the board's hardware resources. Each section above demonstrates the registers and bitwise operations required for setting up ADC, GPIOs, timers, interrupts, EEPROM, and serial communication directly.

### Summary

This approach allows you to maximize efficiency and control, making it ideal for real-time energy monitoring systems like yours. By directly manipulating registers, you maintain precise timing for sensor data sampling, ensure timely responses for alerts, and effectively store data without overhead from standard library functions.