- Comprehensive Project Documentation
- Part 1: Industrial Process Control System
  - 1. System Overview
    - 1.1 Core Features
  - 2. Architecture & Technology Stack
    - 2.1 Core Framework
    - 2.2 UI & Styling
    - 2.3 State Management
  - 3. System Architecture
    - 3.1 Directory Structure
    - 3.2 Component Hierarchy
  - 4. Core Components
    - 4.1 ProcessControlDashboard
    - 4.2 MultiParameterControlDashboard
    - 4.3 ProcessParameterCard
    - 4.4 Visualization Components
      - 4.4.1 VerticalGauge
      - 4.4.2 ConvergenceIndicator
  - 5. Data Flow
    - 5.1 Single Parameter Control Flow
    - 5.2 Multi-Parameter Control Flow
    - 5.3 Real-time Data Simulation
  - 6. User Interface Features
    - 6.1 Responsive Design
    - 6.2 Dark/Light Theme
    - 6.3 Interactive Controls
    - 6.4 Status Visualization
- Part 2: Excel Processor
  - 1. System Overview
    - 1.1 Key Features
  - 2. Data Structure
    - 2.1 Input Format
    - 2.2 Output Format
  - 3. Technical Implementation
    - 3.1 Year Extraction
    - 3.2 Data Processing
    - 3.3 Main Processing Function

- 4. Project Structure
    - 4.1 Files
    - 4.2 Testing
- 5. Usage Example

Comprehensive Project Documentation

This documentation covers two distinct systems:

1. **Industrial Process Control System** - A Next.js-based web application for process control visualization
2. **Excel Processor** - A Python utility for processing dispatcher data from Excel files

---

Part 1: Industrial Process Control System

1. System Overview

The Industrial Process Control System is a modern web application built with Next.js that simulates and visualizes industrial control processes. It demonstrates real-time monitoring and control capabilities with an emphasis on user interface design and real-time data visualization.

1.1 Core Features

- **Two operational modes**: Single Parameter and Multi-Parameter control
- **AI-powered setpoint optimization** with manual override capabilities
- **Real-time data simulation** with realistic process dynamics
- **Modern UI** with responsive design and dark/light theme support
- **Interactive controls** for process parameter adjustment

2. Architecture & Technology Stack

2.1 Core Framework

- **Next.js 15.2.4** - React-based full-stack framework with App Router
- **TypeScript** - Type-safe JavaScript for better code quality
- **React 18** - Component-based UI library with Hooks API

2.2 UI & Styling

- **Tailwind CSS** - Utility-first CSS framework for styling
- **Radix UI** - Collection of accessible component primitives
- **Lucide React** - Icon library for visual elements
- **Recharts** - Composable charting library for data visualization

2.3 State Management

- **React Hooks** - `useState`, `useEffect` for local state management
- **Custom Hooks** - `use-mobile`, `use-toast` for specialized functionality
- **Real-time Simulation** - `setInterval` for continuous data updates

3. System Architecture

3.1 Directory Structure

```
industrial-process-control/
├── app/                          # Next.js App Router pages
│   ├── globals.css               # Global styles
│   ├── layout.tsx                # Root layout with navigation
│   ├── page.tsx                  # Home page (Single Parameter)
│   └── multi-parameter/
│       └── page.tsx              # Multi-parameter page
├── components/                   # React components
│   ├── ui/                       # Reusable UI components (50 files)
│   ├── process-control-dashboard.tsx      # Single parameter dashboard
│   ├── multi-parameter-control-dashboard.tsx  # Multi-parameter dashboard
│   ├── process-parameter-card.tsx         # Individual parameter control card
│   ├── fraction-main-display.tsx          # Fraction separation display
│   ├── vertical-gauge.tsx               # Gauge visualization
│   ├── process-trend-chart.tsx          # Real-time trending charts
│   ├── convergence-indicator.tsx        # AI convergence status
│   └── theme-provider.tsx             # Dark/light theme support
```

```
├── hooks/                    # Custom React hooks
├── lib/                     # Utility libraries
├── styles/                  # Additional styling
├── public/                  # Static assets
└── Configuration files (package.json, tsconfig.json, etc.)
```

3.2 Component Hierarchy

```
RootLayout (app/layout.tsx)
├── Navigation Bar
│   ├── "Process Control System" Title
│   └── Navigation Links
│       ├── "Single Parameter" → /
│       └── "Multi Parameter" → /multi-parameter
├── ThemeProvider (Dark/Light mode)
└── Page Content
    ├── Home (app/page.tsx)
    │   └── ProcessControlDashboard
    └── Multi-Parameter (app/multi-parameter/page.tsx)
        └── MultiParameterControlDashboard
```

4. Core Components

4.1 ProcessControlDashboard

Purpose: Main interface for single-parameter control systems.

Technical Implementation:

```
// components/process-control-dashboard.tsx
interface ProcessData {
  timestamp: number
  pv: number
  sp: number
  aiSp: number
}

export function ProcessControlDashboard() {
  // State variables
  const [processData, setProcessData] = useState<ProcessData[]>([])
  const [currentPV, setCurrentPV] = useState(45.2)
  const [currentSP, setCurrentSP] = useState(50.0)
  const [aiEnabled, setAiEnabled] = useState(true)
  const [spLowLimit, setSpLowLimit] = useState([20])
  const [spHighLimit, setSpHighLimit] = useState([80])
  const [manualSP, setManualSP] = useState(50.0)

  // Simulation logic
  useEffect(() => {
    const interval = setInterval(() => {
      const now = Date.now()
      const aiSp = aiEnabled
        ? Math.max(spLowLimit[0], Math.min(spHighLimit[0], currentSP +
(Math.random() - 0.5) * 2))
        : manualSP

      // Simulate PV trying to follow SP with lag and noise
      const error = aiSp - currentPV
      const newPV = currentPV + error * 0.1 + (Math.random() - 0.5) * 1.5

      setCurrentPV(newPV)
      setCurrentSP(aiSp)

      // Update historical data with sliding window
      setProcessData((prev) => [
        ...prev,
        { timestamp: now, pv: newPV, sp: aiSp, aiSp: aiSp }
      ].slice(-50))
    }, 1000)

    return () => clearInterval(interval)
  }, [currentPV, currentSP, aiEnabled, spLowLimit, spHighLimit, manualSP])

  // Remaining component rendering
  // ...
}
```

Key Features:

- Real-time data simulation at 1-second intervals
- Toggleable AI control mode vs. manual setpoint
- Process variable visualization with trend charts and gauges
- Convergence status monitoring with indicators
- System status cards with performance metrics

4.2 MultiParameterControlDashboard

Purpose: Interface for complex multi-variable process control with parameter interdependencies.

Technical Implementation:

```tsx
// components/multi-parameter-control-dashboard.tsx
interface ProcessParameter {
  id: string
  name: string
  unit: string
  pv: number
  sp: number
  aiSp: number
  lowLimit: number
  highLimit: number
  trend: Array<{ timestamp: number; pv: number; sp: number }>
  color: string
  icon: string
}

export function MultiParameterControlDashboard() {
  // State for 8 different process parameters
  const [parameters, setParameters] = useState<ProcessParameter[]>([
    // Parameter definitions with initial values
    {
      id: "ore",
      name: "Ore Feed Rate",
      unit: "t/h",
      pv: 125.5,
      sp: 130.0,
      aiSp: 130.0,
      lowLimit: 100,
      highLimit: 200,
      trend: [],
      color: "amber",
      icon: "⛏️",
    },
    // Additional parameters...
  ])

  const [fractionData, setFractionData] = useState<FractionData[]>([])
  const [currentFraction, setCurrentFraction] = useState(78.5)
  const [targetFraction, setTargetFraction] = useState(82.0)

  // Simulation with parameter interdependencies
  useEffect(() => {
    const interval = setInterval(() => {
      // Update parameters with cross-coupling effects
      setParameters((prev) => prev.map(param => {
        // AI optimization logic with parameter influence
        // ...
        return updatedParameter
      }))

      // Calculate overall fraction based on parameter influences
      const calculatedFraction = calculateFractionFromParameters()
      setCurrentFraction(calculatedFraction)

      // Update historical data
      setFractionData(prev => [...prev, newDataPoint].slice(-50))
    }, 2000)
    return () => clearInterval(interval)
  }, [parameters, currentFraction, targetFraction])

  // Remaining component rendering
  // ...
}
```

Key Features:

- Management of 8 interconnected process parameters
- Cross-coupling simulation between parameters
- Fraction separation visualization
- Individual parameter cards with mini-trend charts
- System-wide performance metrics

4.3 ProcessParameterCard

Purpose: Encapsulates control and visualization for an individual process parameter.

Technical Implementation:

```
// components/process-parameter-card.tsx
interface ProcessParameterCardProps {
  parameter: ProcessParameter
  onParameterUpdate: (parameter: ProcessParameter) => void
}

export function ProcessParameterCard({ parameter, onParameterUpdate }:
ProcessParameterCardProps) {
  const error = parameter.sp - parameter.pv
  const errorPercentage = Math.abs(error / parameter.sp) * 100
  const isInRange = Math.abs(error) < (parameter.highLimit - parameter.lowLimit) *
0.05

  // Mini-chart data preparation
  const chartData = parameter.trend.map(point => ({
    time: new Date(point.timestamp).toLocaleTimeString(),
    PV: point.pv.toFixed(1),
    SP: point.sp.toFixed(1),
  }))

  return (
    <Card className="shadow-lg border-0...">
      <CardHeader>
        {/* Parameter name and status indicators */}
      </CardHeader>
      <CardContent>
```

```
        {/* Current values display */}
        {/* Mini trend chart */}
        {/* Control elements */}
      </CardContent>
    </Card>
  )
}
```

4.4 Visualization Components

4.4.1 VerticalGauge

Purpose: Visual representation of process variable as a vertical gauge.

```
// components/vertical-gauge.tsx
interface VerticalGaugeProps {
  value: number
  min: number
  max: number
  label: string
  color: "blue" | "green" | "red" | "orange"
  unit?: string
}

export function VerticalGauge({ value, min, max, label, color, unit = "" }:
VerticalGaugeProps) {
  const percentage = Math.max(0, Math.min(100, ((value - min) / (max - min)) *
100))

  return (
    <div className="flex flex-col items-center">
      <div className="relative w-8 h-48 bg-slate-200 dark:bg-slate-700 rounded-full
overflow-hidden">
        {/* Background scale marks */}
        {/* Fill based on percentage */}
        <div
          className={`absolute bottom-0 w-full bg-gradient-to-t
${colorClasses[color]}`}
          style={{ height: `${percentage}%` }}
        />
        {/* Scale labels */}
      </div>
    </div>
  )
}
```

4.4.2 ConvergenceIndicator

Purpose: Displays system convergence status and recommended actions.

```
// components/convergence-indicator.tsx
interface ConvergenceIndicatorProps {
  error: number
  errorPercentage: number
  isConverged: boolean
}

export function ConvergenceIndicator({ error, errorPercentage, isConverged }:
ConvergenceIndicatorProps) {
  const getActionIcon = () => {
    if (isConverged) return <CheckCircle className="h-8 w-8 text-green-600" />
    if (error > 2) return <ArrowUp className="h-8 w-8 text-red-600" />
    if (error < -2) return <ArrowDown className="h-8 w-8 text-blue-600" />
    return <Minus className="h-8 w-8 text-orange-600" />
  }

  // Display convergence status and guidance
  return (
    <div className="space-y-4">
      {/* Status indicators */}
      {/* Performance metrics */}
      {/* Action guidance */}
    </div>
  )
}
```

5. Data Flow

5.1 Single Parameter Control Flow

```
User Input → State Update → AI Processing → Process Simulation → UI Update → User
Input...
```

1. **User Input**: Manual setpoint entry or limits adjustment
2. **State Update**: React state changes via setter functions
3. **AI Processing**: Calculated setpoints within constraints
4. **Process Simulation**: PV follows SP with lag and noise
5. **UI Update**: Visual components reflect new values

5.2 Multi-Parameter Control Flow

```
8 Parameters → Individual Control → Cross-coupling → Fraction Output → Optimization
→ Parameters
```

1. **Parameter Influence**: Each parameter affects the overall process
2. **Cross-Coupling**: Changes in one parameter affect others
3. **Fraction Calculation**: Overall output metric derived from all parameters
4. **AI Optimization**: Adjustments to optimize fraction output
5. **Feedback Loop**: Continuous adjustment based on performance

5.3 Real-time Data Simulation

Key aspects of the data simulation:

1. **Process Dynamics**:

```
// Simplified proportional control with noise
const error = aiSp - currentPV
const newPV = currentPV + error * 0.1 + (Math.random() - 0.5) * 1.5
```

2. **AI Optimization**:

```
// AI adjusts setpoint within constraints
const aiSp = Math.max(spLowLimit[0], Math.min(spHighLimit[0],
  currentSP + (Math.random() - 0.5) * 2))
```

3. **Parameter Interdependence**:

```
// Multi-parameter influence calculation
const oreInfluence = ((parameters[0]?.pv || 125) / 130) * 20
const waterInfluence = ((parameters[1]?.pv || 45) / 48) * 15
// Combined calculation of fraction
const calculatedFraction = oreInfluence + waterInfluence + powerInfluence +
densityInfluence + noise
```

6. User Interface Features

6.1 Responsive Design

- Mobile-first approach with responsive grid layouts
- Adapts from single column to multi-column based on screen size
- Optimized component sizing for different devices

6.2 Dark/Light Theme

- System-aware theme detection
- Theme switching via ThemeProvider
- Consistent styling across theme changes

6.3 Interactive Controls

- Slider controls for setpoints and limits
- Toggle switches for mode selection
- Input fields for precise value entry
- Real-time updates on user interaction

6.4 Status Visualization

- Color-coded status indicators
- Progress bars for performance metrics
- Status badges for system state
- Action recommendations based on current state

Part 2: Excel Processor

1. System Overview

The Excel Processor is a Python utility designed to process dispatcher data from Excel files. It extracts data from multiple sheets representing different months, adds timestamps, and exports the processed data to both CSV and Excel formats.

1.1 Key Features

- Processes Excel files with multiple sheets (one per month)
- Automatically extracts year from filename
- Handles shift-based data with proper timestamps
- Filters out summary rows ("Общо")
- Processes current month data up to current date
- Exports to both CSV and Excel formats

2. Data Structure

2.1 Input Format

- Excel file with 12 sheets (one per month)
- Each sheet contains shift data for each day
- Three shifts per day:
    - Shift 1: 06:00
    - Shift 2: 14:00
    - Shift 3: 22:00

2.2 Output Format

- Processed data with additional columns:
    - TimeStamp: Combined date and shift time
    - OriginalSheet: Source month sheet
- Files named with year from filename:
    - `processed_dispatcher_data_YYYY.csv`
    - `processed_dispatcher_data_YYYY.xlsx`

3. Technical Implementation

3.1 Year Extraction

```python
def extract_year_from_filename(filename):
    """
    Extract 4-digit year pattern (20XX) from filename.
    Falls back to current year if not found.

    Args:
        filename (str): Input filename or path

    Returns:
        int: Extracted year
    """
    # Extract base filename from path if necessary
    base_filename = os.path.basename(filename)

    # Search for 20XX pattern in filename
    match = re.search(r'20\d{2}', base_filename)

    if match:
        return int(match.group())
    else:
        # Fall back to current year
        return datetime.now().year
```

3.2 Data Processing

```python
def process_sheet(sheet_name, sheet_data, year):
    """
    Process a single month sheet with proper timestamps.

    Args:
        sheet_name (str): Name of the sheet (month)
        sheet_data (DataFrame): Sheet data
        year (int): Year for timestamps

    Returns:
        DataFrame: Processed data with timestamps
    """
    # Filter out summary rows
    filtered_data = sheet_data[sheet_data["Смяна"] != "Общо"]

    # Map month name to month number
    month_num = month_mapping.get(sheet_name, 1)  # Default to January

    # Add timestamps based on shift
    filtered_data["TimeStamp"] = filtered_data.apply(
        lambda row: create_timestamp(row, year, month_num),
        axis=1
    )

    # Add original sheet column
    filtered_data["OriginalSheet"] = sheet_name
    return filtered_data
```

3.3 Main Processing Function

```python
def excel_to_dataframe(file_path):
    """
    Process Excel file with multiple sheets.

    Args:
        file_path (str): Path to Excel file

    Returns:
        DataFrame: Combined processed data
    """
    # Extract year from filename
    year = extract_year_from_filename(file_path)

    # Load Excel file
    excel = pd.ExcelFile(file_path)

    # Process each sheet
    all_data = []
    for sheet_name in excel.sheet_names:
        sheet_data = excel.parse(sheet_name)
        processed_data = process_sheet(sheet_name, sheet_data, year)

        # For current month, filter to current date
        if is_current_month(sheet_name):
            processed_data = filter_to_current_date(processed_data)

        all_data.append(processed_data)

    # Combine all processed sheets
    combined_data = pd.concat(all_data, ignore_index=True)

    # Save with year in filename
    combined_data.to_csv(f"processed_dispatcher_data_{year}.csv", index=False)
    combined_data.to_excel(f"processed_dispatcher_data_{year}.xlsx", index=False)

    return combined_data
```

4. Project Structure

4.1 Files

- `dispather_excel_processor.py`: Main module with data processing logic
- `convert_dispatcher_data.py`: Script that demonstrates usage and runs data quality checks
- `test_year_extraction.py`: Test script for filename year extraction

4.2 Testing

The `test_year_extraction.py` file ensures the year extraction function works with various filename formats:

```
def test_year_extraction():
    # Test various filename formats
    assert extract_year_from_filename("Doklad_Dispecheri_2024.xlsx") == 2024
    assert extract_year_from_filename("/path/to/Report_2023_January.xlsx") == 2023
    assert extract_year_from_filename("C:\\Data\\2022\\Dispatcher.xlsx") == 2022

    # Test fallback to current year
    assert extract_year_from_filename("Dispatcher_Report.xlsx") ==
datetime.now().year
```

5. Usage Example

```
from dispather_excel_processor import excel_to_dataframe

# Process the dispatcher Excel file
data = excel_to_dataframe("Doklad_Dispecheri_2024.xlsx")

# Display summary statistics
print(f"Processed {len(data)} records")
print(f"Date range: {data['TimeStamp'].min()} to {data['TimeStamp'].max()}")

# Run data quality checks
check_data_quality(data)
```