ADC-UART Data Acquisition System for OLTC Analysis

Project Overview

This project implements a high-performance data acquisition and analysis system for On-Load Tap Changer (OLTC) with vacuum interrupters, featuring:

- 10kHz ADC sampling of analog signals from Rogowski coil (current measurement)
- GPIO event detection with precise timestamping for arc voltage detection via LED signals
- UART communication for data transmission to a PC
- Real-time visualization and analysis with a PyQt-based GUI
- Phase angle calculation between current and voltage signals
- Arc duration measurement and analysis

System Architecture

STM32G474 Nucleo Firmware

The STM32G474 Nucleo microcontroller firmware handles:

1. **ADC Sampling**
    - 10kHz sampling rate for 3.3V amplitude signals
    - Uses DMA for efficient data transfer without CPU intervention
    - 12-bit resolution (±1.65V) with samples stored in 2 bytes
    - Ring buffer implementation (1024 bytes) for efficient operations
    - Supports both continuous (timer-triggered) and event-triggered sampling modes
    - Optimized for Rogowski coil signals (100mV/kA sensitivity)
2. **GPIO Monitoring**
    - PA1 configured as external interrupt on rising edge
    - Precise timestamping of LED signal pulses (indicating voltage zero-crossings)
    - GPIO state changes stored with 4-byte timestamps
    - Supports interrupt-driven event detection
    - Capture of arc events and re-ignition voltage signals
3. **Data Transmission**

- UART transmission in interrupt mode with DMA support
- Frame format with header (0xAA55) and footer (0x55AA)
- 400 bytes of ADC data (200 samples) per frame
- Support for both debug and data UART interfaces
- Packet-based communication protocol with different packet types:
  - 0xA0: ADC data packets (current measurements)
  - 0xB0: GPIO event packets (voltage zero-crossing events)
  - 0xC0: Timestamp synchronization packets
  - 0xD0: Status/debug packets

4. **DMA Management**

   - Efficient DMA transfers for both ADC data acquisition and UART transmission
   - Error handling and state monitoring for reliable operation
   - Automatic recovery from DMA transfer failures

Python GUI Application

The Python application provides:

1. **Serial Communication**

   - Automatic COM port detection and selection
   - Support for both endianness formats
   - Frame validation and statistics reporting
   - Packet processing with header-based protocol identification
   - Optimized buffer management for continuous data flow

2. **Data Visualization**

   - Real-time plotting of ADC data (current signal) using PyQtGraph
   - GPIO event visualization (voltage signal) with timestamp correlation
   - Signal analysis capabilities including:
     - Zero-crossing detection for both current and voltage
     - Phase angle calculation between signals
     - Arc event detection and duration measurement
     - Signal frequency and amplitude calculation
   - Adjustable time axis and voltage scaling

3. **User Interface**

   - Intuitive controls for starting/stopping data acquisition
   - Mode selection (Continuous vs. Interrupt)

  - Serial port selection and connection management
  - Debug logging with packet inspection capabilities
  - Professionally organized analysis results with clear sections:
    - System Overview & Events (frequencies, amplitudes, timestamps)
    - Phase Analysis & Insights (phase angles with statistics)

4. **Signal Analysis**

  - Current and voltage zero-crossing detection with precise timestamping
  - Phase angle calculation with statistical analysis (mean, min, max)
  - Arc event detection and duration measurement
  - Automated filtering of initial burst signals for accurate calculations
  - Recovery voltage analysis specific to OLTC with vacuum interrupters

MCU Core Structure

The STM32G474 firmware is organized into a standard STM32CubeIDE project structure, with the `Core` folder containing the main application code:

Core Folder Organization

- **Inc/**: Header files
- **Src/**: Source files
- **Startup/**: Processor startup code

Key Files and Components

Configuration and Utilities

- **config.h/config.c**:

  - System-wide configuration parameters
  - Adjustable settings for ADC sampling rates, buffer sizes, and UART configurations
  - Debug mode flags and feature toggles
  - Pin definitions and hardware-specific constants

- **common.h/common.c**:

  - Common utility functions used throughout the application
  - Helper macros and inline functions
  - Data type definitions and conversions
  - Buffer management utilities

- **ringbuffer.h/ringbuffer.c**:
    - Implementation of circular buffer data structure
    - Thread-safe read/write operations for data exchange between ISRs and main context
    - Buffer overflow protection and status reporting
    - Optimized for ADC samples and timestamp storage

Peripheral Drivers

- **madc.h/madc.c** (ADC Manager):
    - ADC peripheral initialization and configuration
    - DMA channel setup for ADC data transfer
    - Sampling mode control (continuous vs. event-triggered)
    - ADC calibration and error handling
    - Data preprocessing and filtering options
- **muart.h/muart.c** (UART Manager):
    - UART peripheral configuration (115200 baud, 8-bit, EVEN parity)
    - TX/RX buffer management
    - DMA-based transmission for efficient data transfer
    - Packet framing and protocol implementation
    - Error detection and recovery mechanisms
- **timer.h/timer.c**:
    - Timer initialization for precise timestamping
    - Microsecond resolution timing functions
    - Timestamp synchronization between GPIO events and ADC data
    - Timebase management for system scheduling

Interrupt Handling

- **handlers.h/handlers.c**:
    - Custom interrupt handlers for GPIO, DMA, and timer events
    - Event prioritization and processing logic
    - Data synchronization between interrupt context and main application
    - Error handling and recovery procedures for interrupt-related failures
    - Timestamp correlation between ADC samples and GPIO events
- **stm32g4xx_it.h/stm32g4xx_it.c**:

- STM32 HAL interrupt handlers and exception vectors
- System exception handlers (HardFault, MemManage, etc.)
- Peripheral interrupt routing to custom handlers

- **callbacks.c**:
  - HAL callback implementations for peripherals
  - DMA transfer complete and half-complete handlers
  - Timer overflow and update callbacks
  - Error handling callbacks for graceful error recovery

System and Main Application

- **main.c/main.h**:
  - Application entry point and main loop
  - System initialization sequence
  - Mode selection handling (continuous vs. interrupt mode)
  - Command processing from UART
  - Overall system state management

- **stm32g4xx_hal_msp.c**:
  - MCU-specific peripheral initialization
  - Clock configuration for optimal performance
  - GPIO, DMA, and interrupt priority configuration
  - HAL MSP (MCU Support Package) initialization callbacks

- **system_stm32g4xx.c**:
  - System clock configuration
  - Core frequency settings (170MHz operation)
  - Flash latency and power settings

Firmware Operation Flow

1. **Initialization Phase**:
   - System clocks are configured for 170MHz operation
   - Peripherals (GPIO, ADC, UART, Timers) are initialized
   - DMA channels are configured for ADC and UART
   - Ring buffers are initialized for data storage

2. **Command Processing**:

- The system waits for commands from the UART interface
   - Commands trigger mode changes, reset operations, or status reporting

3. **Data Acquisition**:

   - In Continuous Mode: ADC samples at 10kHz based on timer triggering
   - In Interrupt Mode: ADC sampling starts on PA1 rising edge
   - DMA transfers ADC data to memory without CPU intervention
   - GPIO events are timestamped and stored in a separate buffer

4. **Data Transmission**:

   - ADC samples are packed into frames (200 samples per frame)
   - GPIO events are transmitted with timestamps
   - UART transmits data using DMA for efficiency
   - Error detection ensures data integrity

5. **Error Handling**:

   - DMA errors are detected and recovery procedures initiated
   - Buffer overflow conditions are monitored and reported
   - System can reset peripherals if persistent errors occur

Operation Modes

Continuous Mode (Timer-Triggered)

- ADC samples are taken at a fixed rate of 10kHz
- Timer3 is used to trigger ADC conversions
- Data is continuously transmitted when 200 samples are collected
- Ideal for periodic signal monitoring and analysis
- **Note:** This mode is currently experiencing some issues where interrupts cause distortion to the ADC signal

Interrupt Mode (Event-Triggered)

- ADC sampling begins when a rising edge is detected on PA1
- Precise timestamping of the trigger event
- Sampling continues until the specified number of samples is collected
- Ideal for capturing transient events and analyzing signals in response to external triggers
- **Recommended:** This mode is working well and provides reliable results

Hardware Requirements

- STM32G474 Nucleo development board
- USB-to-UART converter for PC communication (built into Nucleo board)
- Analog signal source for ADC input (0-3.3V range)
  - Rogowski coil (100mV/kA sensitivity) for current measurement
- Digital signal source (LED sensor) for voltage zero-crossing detection
- Power supply (USB or external 3.3V)
- Optional oscilloscope for signal verification

Software Requirements

- **STM32 Development:**
  - STM32CubeIDE (v1.9.0 or later) for firmware development and flashing
  - STM32CubeMX for peripheral configuration and code generation
  - STM32G4 HAL/LL libraries
- **Python Development:**
  - Visual Studio Code as the primary IDE
  - Python 3.6+ with the following packages:
    - PyQt5 for the GUI framework
    - PyQtGraph for real-time plotting
    - NumPy for data processing
    - pySerial for serial communication
  - VSCode Extensions:
    - Python extension
    - Pylance for intelligent code completion
    - Python Docstring Generator
    - Python Test Explorer

Getting Started

STM32 Development Setup

1. **Install Required Software:**
   - Download and install STM32CubeIDE
   - Download and install STM32CubeMX (if not included in CubeIDE)

2. **STM32G474 Nucleo Configuration:**

    - Connect the STM32G474 Nucleo board to your computer via USB
    - Open STM32CubeMX and create a new project:
        - Select STM32G474RE Nucleo board
        - Configure the clock tree for maximum performance (170MHz)
        - Configure ADC1 for continuous sampling with DMA
        - Configure GPIO PA1 for external interrupt (EXTI1)
        - Configure UART2 for communication with PC (115200 baud)
        - Configure TIM2 for precise timestamping
        - Configure DMA channels for ADC and UART
    - Generate code and open in STM32CubeIDE

3. **Firmware Customization:**

    - Adjust `config.h` for your specific application
    - Implement DMA handlers for ADC and UART
    - Implement EXTI interrupt handlers for GPIO events
    - Implement data packet formatting and transmission routines
    - Build and flash the firmware to the Nucleo board

Python Application Setup in VSCode

1. **Install Visual Studio Code:**

    - Download and install VS Code
    - Install recommended extensions for Python development

2. **Project Setup:**

    - Clone this repository
    - Open the project folder in VS Code
    - Create a Python virtual environment:

    ```
    python -m venv venv
    ```

    - Activate the virtual environment:
        - Windows: `venv\Scripts\activate`
        - Linux/macOS: `source venv/bin/activate`

3. **Install Dependencies:**

```
cd py
pip install -r requirements.txt
```

4. **Configure VSCode:**
    - Select the Python interpreter from your virtual environment
    - Configure the Python extension settings for linting and formatting
    - Set up the integrated terminal to use your virtual environment
5. **Running the Application:**
    - Open the integrated terminal in VSCode
    - Navigate to the `py` directory
    - Activate your virtual environment if not already active
    - Run the application: `python main.py`

Operating Instructions

Hardware Connection

1. Connect the STM32G474 Nucleo board to your PC via USB
2. Connect the Rogowski coil to the ADC input (PA0)
3. Connect the LED sensor output to the GPIO input (PA1)
4. Ensure all signals are within the 0-3.3V range

Starting the Application

1. Launch VS Code and open the project folder
2. Open a terminal and navigate to the `py` directory
3. Activate your virtual environment if not already active
4. Run the application: `python main.py`

Using the GUI

1. **Connection Setup:**
    - Select the appropriate COM port from the dropdown menu
    - Choose the desired operation mode (Continuous or Interrupt)
    - Click "Start" to begin data acquisition and visualization
2. **Real-time Monitoring:**
    - Observe the current signal (blue sine wave) and voltage events (red pulses)

    - The time scale is in milliseconds for precise event timing
    - Data is displayed in real-time with automatic scaling

3. **Analysis:**

    - Zero-crossings are automatically detected and timestamped
    - Phase angles are calculated between corresponding current and voltage events
    - Arc events are detected and their duration is measured
    - Analysis results are organized in the right panels:
        - **System Overview & Events:** Shows system info, signal metrics, timestamps
        - **Phase Analysis & Insights:** Shows phase angles and statistical analysis

4. **Controls:**

    - Start/Stop: Toggle data acquisition
    - Reset: Clear all data and reset the display
    - Mode Selection: Switch between continuous and interrupt modes

OLTC-Specific Analysis

Understanding the OLTC with Vacuum Interrupter

The On-Load Tap Changer (OLTC) with vacuum interrupters has specific characteristics:

- The arcing current goes to zero before the recovery voltage appears
- Re-ignition voltage is a critical parameter for OLTC performance
- Phase angle calculations represent the delay between current zero and recovery voltage

Interpreting the Results

- **Current Zero-Crossings:** Points where the arcing current passes through zero
- **Voltage Zero-Crossings:** Derived from LED signal pulses, representing re-ignition voltage events
- **Phase Angle:** Represents the delay between current extinction and voltage recovery, measured in degrees (relative to 50/60Hz period)
- **Arc Duration:** Time between arc ignition and extinction, critical for OLTC performance evaluation

Advanced Analysis

- The application automatically excludes the initial burst of pulses for accurate phase calculations
- Statistical analysis helps identify variations in phase angles across multiple cycles
- Frequency and amplitude calculations provide insights into current characteristics

Troubleshooting

STM32 Firmware Issues

- **Flashing Problems:**
  - Ensure the Nucleo board is properly connected and recognized
  - Check jumper configurations on the Nucleo board
  - Try using ST-Link Utility for direct flashing
- **DMA Issues:**
  - Verify DMA channel priorities and configurations
  - Check for resource conflicts in CubeMX configuration
  - Monitor transfer complete and error flags in the HAL callbacks
- **Signal Integrity:**
  - Ensure input signals are within 0-3.3V range
  - Use shielded cables for analog signals to reduce noise
  - Add appropriate filtering capacitors for ADC inputs

Python Application Issues

- **Serial Connection Problems:**
  - Verify the Nucleo board appears in Device Manager
  - Check that no other application is using the COM port
  - Try different USB ports or a different USB cable
  - Ensure the correct drivers are installed for the ST-Link Virtual COM port
- **GUI Display Issues:**
  - Update your graphics drivers
  - Check PyQt5 and PyQtGraph versions for compatibility
  - Increase virtual memory if plotting is slow
- **Analysis Errors:**
  - Ensure signal quality is sufficient for accurate zero-crossing detection
  - Adjust the threshold values if zero-crossings are missed
  - Check that both current and voltage signals are present

Performance Optimization

- **STM32 Performance:**
  - Optimize DMA configurations for minimal CPU load
  - Use FIFO modes where appropriate
  - Consider overclocking the STM32G474 for demanding applications
- **Python Application Performance:**
  - Use PyQtGraph's optimization features:
    - Set `antialias=False` for faster plotting
    - Use `skipFiniteCheck=True` for data arrays
  - Adjust the update rate to match your PC's capabilities
  - Consider running with a dedicated GPU if available

Advanced Features

- **Data Saving:** Implement data saving functionality for offline analysis
- **Multiple Channel Support:** Extend the system for multi-channel acquisition
- **Digital Filtering:** Add digital filters for noise reduction
- **Automated Testing:** Configure predefined test sequences
- **Remote Control:** Add network capabilities for remote monitoring

Contributing

Contributions to this project are welcome! Please follow these steps:

1. Fork the repository
2. Create a feature branch
3. Submit a pull request with your changes
4. Ensure all tests pass

License

Acknowledgments

- STM32 HAL library for hardware abstraction
- PyQt and PyQtGraph for the visualization framework
- Contributors and testers who helped improve the system