Software Architecture for Power Converter Firmware

This document outlines the structure and functions required to develop firmware for a power converter system. The firmware is divided into three layers: Driver Layer, Middleware Layer, and Application Layer.

---

1. Driver Layer

The Driver Layer interfaces directly with hardware components. It provides low-level functions to initialize, configure, and control peripherals, abstracting hardware details for higher layers.

Responsibilities:

- Hardware initialization and configuration.
- Direct control of peripherals (e.g., ADC, PWM, GPIO, UART, SPI, CAN).
- Providing interrupt service routines (ISRs) for handling hardware events.

Functions in the Driver Layer:

```c
// Peripheral Drivers
void ADC_Init(void);
void ADC_StartConversion(void);
uint16_t ADC_ReadValue(void);

void PWM_Init(void);
void PWM_SetDutyCycle(float dutyCycle); // Duty cycle range: 0.0 (0%) to 1.0 (100%)
void PWM_Start(void);
void PWM_Stop(void);

void GPIO_Init(void);
void GPIO_SetPin(uint8_t pin, uint8_t state); // State: 0 (LOW), 1 (HIGH)
uint8_t GPIO_ReadPin(uint8_t pin);

// Communication Interfaces
void UART_Init(uint32_t baudRate); // Baud rate in bits per second
void UART_SendData(uint8_t *data, uint16_t length);
void UART_ReceiveData(uint8_t *buffer, uint16_t length);

void SPI_Init(void);
void SPI_Transmit(uint8_t *data, uint16_t length);
void SPI_Receive(uint8_t *buffer, uint16_t length);

void CAN_Init(void); // Initializes the CAN peripheral
void CAN_DeInit(void); // Deinitializes the CAN peripheral

// Interrupt Service Routines
void ISR_ADC(void);
void ISR_PWM(void);
```

```
void ISR_GPIO(void);
```

2. Middleware Layer

The Middleware Layer acts as a bridge between the Driver Layer and the Application Layer. It provides higher-level functionality, such as communication protocols, control algorithms, task scheduling, and data processing.

Responsibilities:

- Implementing communication protocols (e.g., CAN, Modbus).
- Providing control algorithms for system operation (e.g., PFC and DAB control loops).
- Managing task scheduling and real-time operations.
- Processing data (e.g., filtering, buffering).
- Handling system configurations and fault management.

Functions in the Middleware Layer:

```
// Communication Protocols
void CAN_SendMessage(uint32_t id, uint8_t *data, uint8_t length);
void CAN_ReceiveMessage(uint32_t *id, uint8_t *data, uint8_t *length);

void Modbus_Init(void); // Initializes the Modbus protocol
void Modbus_ProcessRequest(uint8_t *request, uint8_t *response);

// Control Algorithms
void PFC_ControlLoop(void); // Totem-Pole PFC control loop
void DAB_ControlLoop(void); // Dual Active Bridge control loop

// Task Scheduler / RTOS
void Scheduler_Init(void);
void Scheduler_AddTask(void (*task)(void), uint32_t interval); // Interval in milliseconds
void Scheduler_Run(void);

// Data Processing
float Filter_Signal(float input); // Example: Low-pass filter
void Buffer_AddData(float data); // Adds data to a circular buffer
float Buffer_GetAverage(void); // Computes the average of buffered data

// System Configurations
void FaultHandler_Init(void);
void FaultHandler_Check(void); // Checks for system faults
void StateMachine_Run(void); // Runs the system state machine
```

Control Algorithms

Control algorithms are used to process system inputs (e.g., sensor data) and generate outputs (e.g., control signals) for various power converter operations. These algorithms are essential for maintaining system stability, efficiency, and performance.

PI Controller:

- Purpose: A Proportional-Integral (PI) controller is used for regulating DC quantities, such as voltage or current, in systems like DC-DC converters or Totem-Pole PFC stages.
- Where It Can Be Used:
  - Regulating the output voltage of a DC-DC converter.
  - Controlling the current in a Totem-Pole PFC stage.

```
typedef struct {
    float kp;       // Proportional gain
    float ki;       // Integral gain
    float integral; // Integral accumulator
    float max;      // Maximum output limit
    float min;      // Minimum output limit
} PI_Controller;

void PI_Controller_Init(PI_Controller *controller, float kp, float ki, float min,
float max);
float PI_Controller_Compute(PI_Controller *controller, float error);
```

PR Controller:

- Purpose: A Proportional-Resonant (PR) controller is used for regulating AC quantities, such as sinusoidal currents or voltages, in systems like inverters or grid-connected converters.
- Where It Can Be Used:
  - Controlling the AC current in a grid-connected inverter.
  - Regulating the output voltage of an AC power supply.

```
typedef struct {
    float kp;       // Proportional gain
    float ki;       // Integral gain
    float kr;       // Resonant gain
    float omega;    // Resonant frequency
    float integral; // Integral accumulator
    float prev_error; // Previous error
} PR_Controller;

void PR_Controller_Init(PR_Controller *controller, float kp, float ki, float kr,
float omega);
float PR_Controller_Compute(PR_Controller *controller, float error);
```

TOGI_PLL:

- Purpose: A Second-Order Generalized Integrator Phase-Locked Loop (TOGI_PLL) is used for estimating the phase and frequency of an AC signal, which is critical for synchronization in grid-connected systems.

- Where It Can Be Used:
  - Synchronizing a grid-connected inverter with the AC grid.
  - Estimating the phase and frequency of an AC signal in power systems.

```
typedef struct {
    float kp;       // Proportional gain
    float ki;       // Integral gain
    float frequency; // Estimated frequency
    float phase;    // Estimated phase
    float integral; // Integral accumulator
} TOGI_PLL;

void TOGI_PLL_Init(TOGI_PLL *pll, float kp, float ki);
void TOGI_PLL_Update(TOGI_PLL *pll, float input, float dt);
```

3. Application Layer

The Application Layer handles system-level initialization, user interfaces, and performance monitoring. It interacts with the Middleware Layer to execute system-level tasks and provide a user-friendly interface.

Responsibilities:

- System initialization and startup.
- Managing user interfaces (e.g., displaying messages, reading inputs).
- Setting and monitoring system modes.
- Logging and displaying performance metrics.
- Providing APIs for external control and parameter management.

Functions in the Application Layer:

```
// System Initialization
void System_Init(void); // Initializes the entire system
void System_Start(void); // Starts the system operation

// User Interfaces
void UI_DisplayMessage(const char *message); // Displays a message to the user
void UI_ReadInput(uint8_t *input); // Reads user input (e.g., button press)

// System Modes
void System_SetMode(uint8_t mode); // Sets the system mode
uint8_t System_GetMode(void); // Gets the current system mode

// Performance Monitoring
void Monitor_LogData(void); // Logs system performance data
void Monitor_DisplayPerformance(void); // Displays performance metrics

// APIs for External Control
void API_SetParameter(uint8_t paramId, float value); // Sets a parameter value
float API_GetParameter(uint8_t paramId); // Gets a parameter value
```

How These Algorithms Fit in the System

- Driver Layer: Provides raw data from sensors (e.g., ADC values) and actuates hardware (e.g., PWM signals).
- Middleware Layer: Processes the raw data using control algorithms (e.g., PI, PR, TOGI_PLL) to generate control signals.
- Application Layer: Uses the control signals to implement system-level behavior (e.g., regulating output voltage, synchronizing with the grid).

Summary of Layer Responsibilities

Layer	Responsibilities
Driver Layer	Direct hardware interaction, peripheral initialization, and interrupt handling.
Middleware Layer	Protocols, control algorithms, task scheduling, data processing, and fault handling.
Application Layer	System-level behavior, user interfaces, performance monitoring, and external APIs.

Header File Description: `My_Controller.h`

This header file defines the implementation of **PI (Proportional-Integral)** and **PR (Proportional-Resonant)** controllers. These controllers are widely used in control systems for regulating DC and AC signals, respectively. The file includes initialization, computation, and reset functions for both controllers.

---

1. PI Controller

The **PI Controller** is used for regulating DC quantities, such as voltage or current, in systems like DC-DC converters or Totem-Pole PFC stages.

Structure: `PI_Params_Typedef`

This structure holds the parameters and state variables for the PI controller.

- **Fields**:
  - `kp`: Proportional gain.
  - `ki`: Integral gain.
  - `Ts`: Sampling time (time interval between successive controller updates).
  - `integral`: Integral term accumulator.
  - `output`: Current output of the PI controller.
  - `output_min`: Minimum output limit.
  - `output_max`: Maximum output limit.
  - `reset_flag`: Reset flag (1 = reset integral term, 0 = normal operation).

Functions

1. `PI_Params_Init`:

- Initializes the `PI_Params_Typedef` structure with the specified parameters.
- **Parameters**:
  - `PI_Params`: Pointer to the `PI_Params_Typedef` structure.
  - `kp`: Proportional gain.
  - `ki`: Integral gain.
  - `Ts`: Sampling time.
  - `output_min`: Minimum output limit.
  - `output_max`: Maximum output limit.

2. **`PIController`**:

   - Computes the PI control output based on the given error and updates the controller state.
   - **Parameters**:
     - `PI_Params`: Pointer to the `PI_Params_Typedef` structure.
     - `error`: The error signal (setpoint - measured value).
     - `Ts`: Sampling time.
   - **Returns**: The computed control output.

3. **`PIController_Reset`**:

   - Resets the integral term of the PI controller by setting it to zero.
   - **Parameters**:
     - `PI_Params`: Pointer to the `PI_Params_Typedef` structure.

---

2. PR Controller

The **PR Controller** is used for regulating AC quantities, such as sinusoidal currents or voltages, in systems like inverters or grid-connected converters.

Structure: `PR_Params_Typedef`

This structure holds the parameters and state variables for the PR controller.

- **Fields**:
  - `kpr`: Proportional gain.
  - `kir`: Resonant gain.
  - `wc`: Bandwidth of the resonant controller.
  - `w0`: Resonant frequency (e.g., grid frequency in radians per second).
  - `u1`, `u2`: Internal state variables for the resonant controller.
  - `Ts`: Sampling time.
  - `output`: Current output of the PR controller.
  - `output_min`: Minimum output limit.
  - `output_max`: Maximum output limit.
  - `k`: Precomputed gain factor for the resonant controller.
  - `W0Ts`: Precomputed term for the resonant frequency and sampling time.
  - `reset_flag`: Reset flag (1 = reset internal states, 0 = normal operation).

Functions

1. **`PR_Params_Init`**:
   - Initializes the `PR_Params_Typedef` structure with the specified parameters.
   - **Parameters**:
     - `PR_Params`: Pointer to the `PR_Params_Typedef` structure.
     - `kpr`: Proportional gain.
     - `kir`: Resonant gain.
     - `Ts`: Sampling time.
     - `W0`: Resonant frequency.
     - `Wc`: Bandwidth of the resonant controller.
     - `output_max`: Maximum output limit.
     - `output_min`: Minimum output limit.
2. **`PRController`**:
   - Computes the PR control output based on the measured signal and updates the controller state.
   - **Parameters**:
     - `PR_Params`: Pointer to the `PR_Params_Typedef` structure.
     - `measure`: The measured signal (e.g., AC current or voltage).
     - `Ts`: Sampling time.
   - **Returns**: The computed control output.
3. **`PRController_Reset`**:
   - Resets the internal states of the PR controller by setting them to zero.
   - **Parameters**:
     - `PR_Params`: Pointer to the `PR_Params_Typedef` structure.

Usage

PI Controller

1. **Initialization**: Use `PI_Params_Init` to configure the PI controller parameters.
2. **Computation**: Call `PIController` in a control loop to compute the control output based on the error signal.
3. **Reset**: Use `PIController_Reset` to reset the integral term when needed (e.g., during system restarts).

PR Controller

1. **Initialization**: Use `PR_Params_Init` to configure the PR controller parameters.
2. **Computation**: Call `PRController` in a control loop to compute the control output based on the measured signal.
3. **Reset**: Use `PRController_Reset` to reset the internal states when needed (e.g., during system restarts).

Applications

- **PI Controller**:

- Regulating DC quantities like voltage or current in DC-DC converters or Totem-Pole PFC stages.
- **PR Controller**:
  - Controlling AC quantities like sinusoidal currents or voltages in grid-connected inverters or AC power supplies.

This header file provides a modular and reusable implementation of PI and PR controllers for various control system applications.

TOGI_PLL Code Description

This code implements a **Second-Order Generalized Integrator Phase-Locked Loop (TOGI_PLL)**, which is used to estimate the phase (`phi`) and frequency (`omega`) of an AC signal. The TOGI_PLL is particularly useful in grid-connected systems for synchronization with the grid voltage.

---

1. Structure: `TOGI_PLL_Params_Typedef`

The `TOGI_PLL_Params_Typedef` structure holds the parameters and state variables required for the TOGI_PLL algorithm.

Fields

- `u1`, `u2`, `u3`: Internal state variables for the TOGI integrator.
- `k`: Gain factor for the TOGI integrator.
- `omega`: Estimated frequency of the AC signal.
- `kp`: Proportional gain for the PLL controller.
- `ki`: Integral gain for the PLL controller.
- `vd`: Voltage in the direct axis (used for error computation).
- `phi`: Estimated phase of the AC signal.
- `I_Term`: Integral term accumulator for the PLL controller.

---

2. Global Variables

- `omega_save`: Stores the last computed frequency (`omega`) for debugging or monitoring purposes.
- `phi_save`: Stores the last computed phase (`phi`) for debugging or monitoring purposes.
- `u1_save`: Stores the last value of the internal state variable `u1` for debugging or monitoring purposes.

---

3. Functions

3.1 `TOGI_PLL_Params_Init`

This function initializes the `TOGI_PLL_Params_Typedef` structure with default values.

Purpose

- Resets all internal state variables (`u1`, `u2`, `u3`) to zero.
- Sets the proportional (`kp`) and integral (`ki`) gains for the PLL controller.

- Initializes the phase (`phi`), frequency (`omega`), and other parameters to their default values.

Input Parameters

- `TOGI_PLL_Params`: Pointer to the `TOGI_PLL_Params_Typedef` structure to initialize.

Code Behavior

- Sets `u1`, `u2`, `u3`, `phi`, and `vd` to `0.0f`.
- Assigns predefined constants (`KP_TOGI`, `KI_TOGI`, `K_TOGI`) to the respective fields.
- Initializes the integral term (`I_Term`) and frequency (`omega`) to `0.0f`.

3.2 `TOGI_PLL`

This function implements the TOGI_PLL algorithm to estimate the phase (`phi`) and frequency (`omega`) of the input AC signal.

Purpose

- Tracks the phase and frequency of an AC signal (`vac_sens`) using a second-order generalized integrator.
- Computes the direct-axis voltage (`vd`) and uses a proportional-integral (PI) controller to adjust the frequency (`omega`).

Input Parameters

- `TOGI_PLL_Params`: Pointer to the `TOGI_PLL_Params_Typedef` structure containing the PLL parameters and state variables.
- `vac_sens`: The sensed AC voltage signal.
- `Ts`: Sampling time (time interval between successive updates).

Code Behavior

1. **Error Calculation**:
    - Computes the error between the input signal (`vac_sens`) and the internal state variable `u1`.
2. **TOGI Integrator Update**:
    - Updates the internal state variables (`u1`, `u2`, `u3`) using the TOGI equations:
        - `u2` is updated based on `u1` and the resonant frequency (`W0Ts`).
        - `u1` is updated using the error, gain factor (`k`), and `u2`.
        - `u3` is updated similarly to `u1`.
3. **Direct-Axis Voltage (`vd`) Calculation**:
    - Computes `vd` using the estimated phase (`phi`) and the internal state variables (`v_alpha` and `v_beta`).
4. **PI Controller**:

- Computes the proportional term (`P_Term`) and updates the integral term (`I_Term`) based on `vd`.
- Updates the frequency (`omega`) using the PI controller output and the nominal frequency (`W0`).

5. **Phase Update**:
   - Updates the phase (`phi`) using the estimated frequency and sampling time.
   - Resets `phi` to `0.0f` if it exceeds `2π` or `-2π`.

6. **Debugging Variables**:
   - Stores the computed `omega`, `phi`, and `u1` in global variables (`omega_save`, `phi_save`, `u1_save`) for debugging or monitoring purposes.

---

4. Key Equations

1. **TOGI Integrator Update**:
   - `u2 += u1 * W0Ts`
   - `u1 += (error * k - u2) * W0Ts`
   - `u3 += (error * k - u3) * W0Ts`

2. **Direct-Axis Voltage (`vd`)**:
   - `vd = cos(phi) * v_alpha + sin(phi) * v_beta`

3. **PI Controller**:
   - `P_Term = vd * kp`
   - `I_Term += vd * ki * Ts`
   - `omega = P_Term + I_Term + W0`

4. **Phase Update**:
   - `phi += omega * Ts`

---

5. Applications

The TOGI_PLL algorithm is commonly used in grid-connected systems for:

- **Grid Synchronization**:
  - Estimating the phase and frequency of the grid voltage for synchronizing inverters.
- **Power Quality Monitoring**:
  - Tracking the phase and frequency of AC signals for power system analysis.
- **Control of Grid-Tied Inverters**:
  - Ensuring proper synchronization of inverters with the grid to avoid phase mismatches.

---

6. Example Usage

```c
#include "TOGI.h"

int main() {
    TOGI_PLL_Params_Typedef pll_params;

    // Initialize the TOGI_PLL parameters
    TOGI_PLL_Params_Init(&pll_params);

    // Simulated AC voltage signal
    float vac_sens = 230.0f; // Example input signal
    float Ts = 0.001f;       // Sampling time (1 ms)

    // Run the TOGI_PLL algorithm
    TOGI_PLL(&pll_params, vac_sens, Ts);

    // Print the estimated phase and frequency
    printf("Estimated Frequency: %f\n", pll_params.omega);
    printf("Estimated Phase: %f\n", pll_params.phi);

    return 0;
}
```