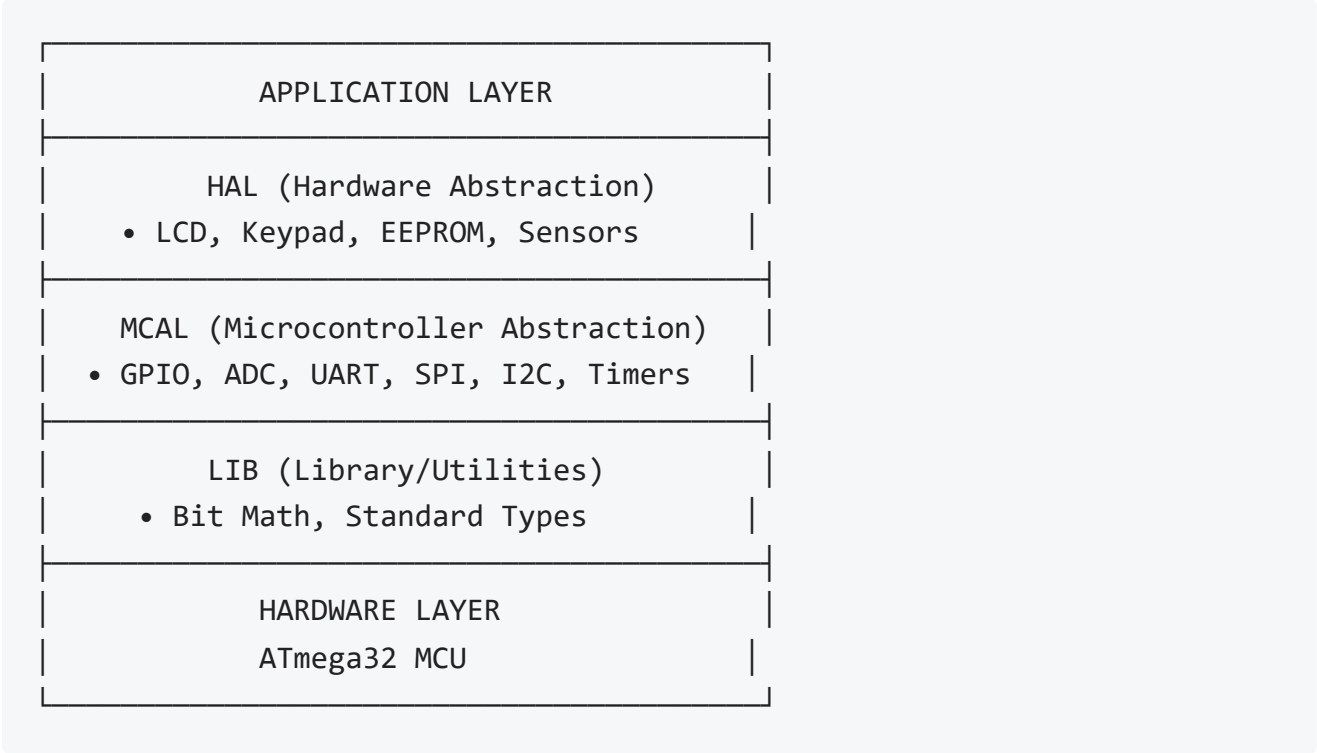
Atmega32_driver_guide

ATmega32 Driver Development Guide - HAL/MCAL Architecture

Table of Contents

Driver Architecture Overview

Layered Architecture Benefits



Architecture Principles:

- **Modularity**: Each driver is self-contained
- **Abstraction**: Higher layers don't need hardware knowledge

- Reusability: Drivers can be reused across projects
- Maintainability: Changes isolated to specific layers
- Portability: Easy to adapt to different microcontrollers

Library Layer (LIB)

1. Standard Types (STD_TYPES.h)

```
#ifndef STD_TYPES_H_
#define STD_TYPES_H_

/* Boolean Data Type */
typedef unsigned char boolean;

/* Boolean Values */
#ifndef FALSE
#define FALSE       (0u)
#endif
#ifndef TRUE
#define TRUE        (1u)
#endif

#define HIGH        (1u)
#define LOW         (0u)

#define NULL_PTR    ((void*)0)

/* Standard Integer Types */
typedef unsigned char         uint8_t;    /*           0 .. 255
*/
typedef signed char           sint8_t;    /*        -128 .. +127
*/
typedef unsigned short        uint16_t;   /*           0 .. 65535
*/
typedef signed short          sint16_t;   /*      -32768 .. +32767
*/
typedef unsigned long         uint32_t;   /*           0 .. 4294967295
*/
typedef signed long           sint32_t;   /* -2147483648 .. +2147483647
*/
typedef unsigned long long    uint64_t;   /*
0..18446744073709551615  */
typedef signed long long      sint64_t;   /*
```

```
-9223372036854775808..9223372036854775807 */
typedef float                   float32_t;
typedef double                  float64_t;

#endif /* STD_TYPES_H_ */
```

2. Bit Math Library (BIT_MATH.h)

```
#ifndef BIT_MATH_H_
#define BIT_MATH_H_

/* Bit Manipulation Macros */
#define SET_BIT(REG, BIT)         (REG |= (1 << BIT))
#define CLR_BIT(REG, BIT)         (REG &= ~(1 << BIT))
#define TOG_BIT(REG, BIT)         (REG ^= (1 << BIT))
#define GET_BIT(REG, BIT)         ((REG >> BIT) & 1)

/* Multi-bit Manipulation */
#define SET_MASK(REG, MASK)       (REG |= MASK)
#define CLR_MASK(REG, MASK)       (REG &= ~MASK)
#define TOG_MASK(REG, MASK)       (REG ^= MASK)
#define GET_MASK(REG, MASK)       (REG & MASK)

/* Nibble Operations */
#define SET_HIGH_NIB(REG, VALUE)  (REG = (REG & 0x0F) | (VALUE << 4))
#define SET_LOW_NIB(REG, VALUE)   (REG = (REG & 0xF0) | (VALUE & 0x0F))
#define GET_HIGH_NIB(REG)         ((REG & 0xF0) >> 4)
#define GET_LOW_NIB(REG)          (REG & 0x0F)

/* Circular Shift Operations */
#define ROL(REG, NUM)             (REG = (REG << NUM) | (REG >> (8 -
NUM)))
#define ROR(REG, NUM)             (REG = (REG >> NUM) | (REG << (8 -
NUM)))

/* Check if bit is set/clear */
#define IS_BIT_SET(REG, BIT)      (REG & (1 << BIT))
#define IS_BIT_CLR(REG, BIT)      (!(REG & (1 << BIT)))

#endif /* BIT_MATH_H_ */
```

MCAL Layer Implementation

1. GPIO Driver (DIO)

DIO_interface.h:

```
#ifndef DIO_INTERFACE_H_
#define DIO_INTERFACE_H_

#include "STD_TYPES.h"

/* Port Definitions */
typedef enum {
    DIO_PORTA = 0,
    DIO_PORTB,
    DIO_PORTC,
    DIO_PORTD
} DIO_Port_t;

/* Pin Definitions */
typedef enum {
    DIO_PIN0 = 0,
    DIO_PIN1, DIO_PIN2, DIO_PIN3,
    DIO_PIN4, DIO_PIN5, DIO_PIN6, DIO_PIN7
} DIO_Pin_t;

/* Direction Definitions */
typedef enum {
    DIO_PIN_INPUT = 0,
    DIO_PIN_OUTPUT
} DIO_PinDirection_t;

/* Value Definitions */
typedef enum {
    DIO_PIN_LOW = 0,
    DIO_PIN_HIGH
} DIO_PinValue_t;

/* Function Prototypes */
void DIO_voidSetPinDirection(DIO_Port_t Copy_Port, DIO_Pin_t Copy_Pin,
DIO_PinDirection_t Copy_Direction);
void DIO_voidSetPinValue(DIO_Port_t Copy_Port, DIO_Pin_t Copy_Pin,
DIO_PinValue_t Copy_Value);
uint8_t DIO_u8GetPinValue(DIO_Port_t Copy_Port, DIO_Pin_t Copy_Pin);
void DIO_voidTogglePinValue(DIO_Port_t Copy_Port, DIO_Pin_t Copy_Pin);
```

```
void DIO_voidSetPortDirection(DIO_Port_t Copy_Port, uint8_t
Copy_Direction);
void DIO_voidSetPortValue(DIO_Port_t Copy_Port, uint8_t Copy_Value);
uint8_t DIO_u8GetPortValue(DIO_Port_t Copy_Port);

#endif /* DIO_INTERFACE_H_ */
```

DIO_program.c:

```
#include "STD_TYPES.h"
#include "BIT_MATH.h"
#include "DIO_interface.h"
#include "DIO_private.h"

void DIO_voidSetPinDirection(DIO_Port_t Copy_Port, DIO_Pin_t Copy_Pin,
DIO_PinDirection_t Copy_Direction) {
    if (Copy_Pin <= DIO_PIN7) {
        switch (Copy_Port) {
            case DIO_PORTA:
                if (Copy_Direction == DIO_PIN_OUTPUT) {
                    SET_BIT(DDRA, Copy_Pin);
                } else {
                    CLR_BIT(DDRA, Copy_Pin);
                }
                break;
            case DIO_PORTB:
                if (Copy_Direction == DIO_PIN_OUTPUT) {
                    SET_BIT(DDRB, Copy_Pin);
                } else {
                    CLR_BIT(DDRB, Copy_Pin);
                }
                break;
            case DIO_PORTC:
                if (Copy_Direction == DIO_PIN_OUTPUT) {
                    SET_BIT(DDRC, Copy_Pin);
                } else {
                    CLR_BIT(DDRC, Copy_Pin);
                }
                break;
            case DIO_PORTD:
                if (Copy_Direction == DIO_PIN_OUTPUT) {
                    SET_BIT(DDRD, Copy_Pin);
                } else {
```

```
                    CLR_BIT(DDRD, Copy_Pin);
                }
                break;
        }
    }
}

void DIO_voidSetPinValue(DIO_Port_t Copy_Port, DIO_Pin_t Copy_Pin,
DIO_PinValue_t Copy_Value) {
    if (Copy_Pin <= DIO_PIN7) {
        switch (Copy_Port) {
            case DIO_PORTA:
                if (Copy_Value == DIO_PIN_HIGH) {
                    SET_BIT(PORTA, Copy_Pin);
                } else {
                    CLR_BIT(PORTA, Copy_Pin);
                }
                break;
            case DIO_PORTB:
                if (Copy_Value == DIO_PIN_HIGH) {
                    SET_BIT(PORTB, Copy_Pin);
                } else {
                    CLR_BIT(PORTB, Copy_Pin);
                }
                break;
            case DIO_PORTC:
                if (Copy_Value == DIO_PIN_HIGH) {
                    SET_BIT(PORTC, Copy_Pin);
                } else {
                    CLR_BIT(PORTC, Copy_Pin);
                }
                break;
            case DIO_PORTD:
                if (Copy_Value == DIO_PIN_HIGH) {
                    SET_BIT(PORTD, Copy_Pin);
                } else {
                    CLR_BIT(PORTD, Copy_Pin);
                }
                break;
        }
    }
}

uint8_t DIO_u8GetPinValue(DIO_Port_t Copy_Port, DIO_Pin_t Copy_Pin) {
```

```
    uint8_t LOC_u8Result = 0;
    if (Copy_Pin <= DIO_PIN7) {
        switch (Copy_Port) {
            case DIO_PORTA:
                LOC_u8Result = GET_BIT(PINA, Copy_Pin);
                break;
            case DIO_PORTB:
                LOC_u8Result = GET_BIT(PINB, Copy_Pin);
                break;
            case DIO_PORTC:
                LOC_u8Result = GET_BIT(PINC, Copy_Pin);
                break;
            case DIO_PORTD:
                LOC_u8Result = GET_BIT(PIND, Copy_Pin);
                break;
        }
    }
    return LOC_u8Result;
}
```

2. ADC Driver Template

ADC_interface.h:

```
#ifndef ADC_INTERFACE_H_
#define ADC_INTERFACE_H_

#include "STD_TYPES.h"

/* ADC Channel Selection */
typedef enum {
    ADC_CHANNEL_0 = 0,
    ADC_CHANNEL_1, ADC_CHANNEL_2, ADC_CHANNEL_3,
    ADC_CHANNEL_4, ADC_CHANNEL_5, ADC_CHANNEL_6, ADC_CHANNEL_7
} ADC_Channel_t;

/* ADC Reference Voltage */
typedef enum {
    ADC_AREF = 0,
    ADC_AVCC,
    ADC_INTERNAL_VREF
} ADC_VoltageReference_t;

/* ADC Prescaler */
```

```c
typedef enum {
    ADC_PRESCALER_2 = 0,
    ADC_PRESCALER_4,
    ADC_PRESCALER_8,
    ADC_PRESCALER_16,
    ADC_PRESCALER_32,
    ADC_PRESCALER_64,
    ADC_PRESCALER_128
} ADC_Prescaler_t;

/* Function Prototypes */
void ADC_voidInit(ADC_VoltageReference_t Copy_VoltageRef,
ADC_Prescaler_t Copy_Prescaler);
uint16_t ADC_u16StartConversionSynch(ADC_Channel_t Copy_Channel);
void ADC_voidStartConversionAsynch(ADC_Channel_t Copy_Channel, void
(*Copy_ptr)(uint16_t));
void ADC_voidDisable(void);

#endif /* ADC_INTERFACE_H_ */
```

3. Timer Driver Template

TIMER_interface.h:

```c
#ifndef TIMER_INTERFACE_H_
#define TIMER_INTERFACE_H_

#include "STD_TYPES.h"

/* Timer Selection */
typedef enum {
    TIMER0 = 0,
    TIMER1,
    TIMER2
} TIMER_Number_t;

/* Timer Modes */
typedef enum {
    TIMER_NORMAL_MODE = 0,
    TIMER_CTC_MODE,
    TIMER_FAST_PWM_MODE,
    TIMER_PHASE_CORRECT_PWM_MODE
} TIMER_Mode_t;
```

```c
/* Timer Prescaler */
typedef enum {
    TIMER_NO_PRESCALING = 1,
    TIMER_PRESCALER_8,
    TIMER_PRESCALER_64,
    TIMER_PRESCALER_256,
    TIMER_PRESCALER_1024
} TIMER_Prescaler_t;

/* Function Prototypes */
void TIMER_voidInit(TIMER_Number_t Copy_TimerNum, TIMER_Mode_t
Copy_Mode, TIMER_Prescaler_t Copy_Prescaler);
void TIMER_voidStart(TIMER_Number_t Copy_TimerNum);
void TIMER_voidStop(TIMER_Number_t Copy_TimerNum);
void TIMER_voidSetCompareValue(TIMER_Number_t Copy_TimerNum, uint16_t
Copy_Value);
void TIMER_voidSetPWMDutyCycle(TIMER_Number_t Copy_TimerNum, uint8_t
Copy_DutyCycle);
void TIMER_voidSetCallBack(TIMER_Number_t Copy_TimerNum, void
(*Copy_ptr)(void));

#endif /* TIMER_INTERFACE_H_ */
```

HAL Layer Implementation

1. LCD Driver Template

LCD_interface.h:

```c
#ifndef LCD_INTERFACE_H_
#define LCD_INTERFACE_H_

#include "STD_TYPES.h"

/* LCD Modes */
typedef enum {
    LCD_4BIT_MODE = 4,
    LCD_8BIT_MODE = 8
} LCD_Mode_t;

/* LCD Configuration Structure */
typedef struct {
    LCD_Mode_t Mode;
    DIO_Port_t DataPort;
```

```
    DIO_Port_t ControlPort;
    DIO_Pin_t RS_Pin;
    DIO_Pin_t EN_Pin;
} LCD_Config_t;

/* Function Prototypes */
void LCD_voidInit(LCD_Config_t* Copy_Config);
void LCD_voidSendCommand(uint8_t Copy_Command);
void LCD_voidSendData(uint8_t Copy_Data);
void LCD_voidSendString(char* Copy_String);
void LCD_voidGoToXY(uint8_t Copy_Row, uint8_t Copy_Col);
void LCD_voidDisplayNumber(sint32_t Copy_Number);
void LCD_voidCreateCustomChar(uint8_t Copy_CharCode, uint8_t*
Copy_CharPattern);
void LCD_voidClearScreen(void);

#endif /* LCD_INTERFACE_H_ */
```

2. Keypad Driver Template

KEYPAD_interface.h:

```
#ifndef KEYPAD_INTERFACE_H_
#define KEYPAD_INTERFACE_H_

#include "STD_TYPES.h"

/* Keypad Configuration */
typedef struct {
    DIO_Port_t RowPort;
    DIO_Port_t ColPort;
    DIO_Pin_t RowPins[4];
    DIO_Pin_t ColPins[4];
    uint8_t KeyMap[4][4];
} KEYPAD_Config_t;

/* Function Prototypes */
void KEYPAD_voidInit(KEYPAD_Config_t* Copy_Config);
uint8_t KEYPAD_u8GetPressedKey(void);
boolean KEYPAD_boolIsKeyPressed(void);

#endif /* KEYPAD_INTERFACE_H_ */
```

File Structure Organization

```
Project_Name/
├── LIB/
│   ├── STD_TYPES.h
│   ├── BIT_MATH.h
│   └── COMMON_MACROS.h
├── MCAL/
│   ├── DIO/
│   │   ├── DIO_interface.h
│   │   ├── DIO_private.h
│   │   ├── DIO_config.h
│   │   └── DIO_program.c
│   ├── ADC/
│   │   ├── ADC_interface.h
│   │   ├── ADC_private.h
│   │   ├── ADC_config.h
│   │   └── ADC_program.c
│   ├── UART/
│   ├── SPI/
│   ├── I2C/
│   ├── TIMER/
│   ├── EXT_INT/
│   └── GIE/
├── HAL/
│   ├── LCD/
│   │   ├── LCD_interface.h
│   │   ├── LCD_private.h
│   │   ├── LCD_config.h
│   │   └── LCD_program.c
│   ├── KEYPAD/
│   ├── EEPROM/
│   ├── STEPPER/
│   └── SERVO/
├── APP/
│   └── main.c
└── Documentation/
    ├── Driver_APIs.md
    └── Hardware_Configuration.md
```

Development Best Practices

1. Header File Structure

```c
#ifndef MODULE_INTERFACE_H_
#define MODULE_INTERFACE_H_

/* Include Dependencies */
#include "STD_TYPES.h"

/* Public Macros */
#define MODULE_MAX_VALUE    255

/* Public Data Types */
typedef enum {
    MODULE_STATE_IDLE = 0,
    MODULE_STATE_BUSY
} MODULE_State_t;

/* Function Prototypes */
void MODULE_voidInit(void);
uint8_t MODULE_u8GetStatus(void);

#endif /* MODULE_INTERFACE_H_ */
```

2. Configuration Management

MODULE_config.h:

```c
#ifndef MODULE_CONFIG_H_
#define MODULE_CONFIG_H_

/* User Configurations */
#define MODULE_ENABLE_INTERRUPT    ENABLE
#define MODULE_TIMEOUT_VALUE       1000
#define MODULE_DEFAULT_MODE        MODULE_FAST_MODE

#endif /* MODULE_CONFIG_H_ */
```

3. Error Handling

```c
typedef enum {
    MODULE_OK = 0,
    MODULE_ERROR,
    MODULE_TIMEOUT,
    MODULE_INVALID_PARAM,
    MODULE_NOT_INITIALIZED
```

```
} MODULE_ErrorStatus_t;

MODULE_ErrorStatus_t MODULE_Init(MODULE_Config_t* Config) {
    if (Config == NULL_PTR) {
        return MODULE_INVALID_PARAM;
    }

    /* Initialize module */

    return MODULE_OK;
}
```

4. Callback Implementation

```
/* Callback Function Type */
typedef void (*MODULE_CallbackFunction_t)(uint8_t);

/* Private Variables */
static MODULE_CallbackFunction_t Global_CallbackPtr = NULL_PTR;

/* Set Callback Function */
void MODULE_voidSetCallback(MODULE_CallbackFunction_t Copy_CallbackPtr)
{
    Global_CallbackPtr = Copy_CallbackPtr;
}

/* ISR Implementation */
ISR(MODULE_vect) {
    if (Global_CallbackPtr != NULL_PTR) {
        Global_CallbackPtr(MODULE_GetData());
    }
}
```

Testing and Validation

1. Unit Testing Template

```
/* Test Framework for Driver Validation */
#include "TEST_FRAMEWORK.h"
#include "DIO_interface.h"

void TEST_DIO_SetPinDirection(void) {
    /* Test Case 1: Valid parameters */
```

```
    DIO_voidSetPinDirection(DIO_PORTA, DIO_PIN0, DIO_PIN_OUTPUT);
    TEST_ASSERT_EQUAL(1, GET_BIT(DDRA, 0));

    /* Test Case 2: Input direction */
    DIO_voidSetPinDirection(DIO_PORTA, DIO_PIN1, DIO_PIN_INPUT);
    TEST_ASSERT_EQUAL(0, GET_BIT(DDRA, 1));

    /* Test Case 3: Invalid pin (boundary test) */
    DIO_voidSetPinDirection(DIO_PORTA, 8, DIO_PIN_OUTPUT);
    /* Should not crash or modify register */

    TEST_PASS("DIO_SetPinDirection tests completed");
}

void TEST_DIO_RunAllTests(void) {
    TEST_DIO_SetPinDirection();
    /* Add more test functions */
}
```

2. Integration Testing

```
/* Integration Test Example */
void TEST_LCD_KEYPAD_Integration(void) {
    LCD_Config_t lcd_config = {
        .Mode = LCD_4BIT_MODE,
        .DataPort = DIO_PORTC,
        .ControlPort = DIO_PORTD,
        .RS_Pin = DIO_PIN0,
        .EN_Pin = DIO_PIN1
    };

    KEYPAD_Config_t keypad_config = {
        .RowPort = DIO_PORTA,
        .ColPort = DIO_PORTB,
        .RowPins = {DIO_PIN0, DIO_PIN1, DIO_PIN2, DIO_PIN3},
        .ColPins = {DIO_PIN0, DIO_PIN1, DIO_PIN2, DIO_PIN3}
    };

    /* Initialize modules */
    LCD_voidInit(&lcd_config);
    KEYPAD_voidInit(&keypad_config);

    /* Test integration */
    LCD_voidSendString("Press Key:");
```

```
    uint8_t key = KEYPAD_u8GetPressedKey();
    LCD_voidSendData(key);

    TEST_PASS("LCD-Keypad integration test passed");
}
```

3. Performance Benchmarking

```
/* Performance Testing Macros */
#define BENCHMARK_START()   TCNT0 = 0; TCCR0 = 0x01
#define BENCHMARK_END()     benchmark_cycles = TCNT0; TCCR0 = 0x00
#define CYCLES_TO_US(cycles) ((cycles * 1000000UL) / F_CPU)

void BENCHMARK_DIO_Operations(void) {
    volatile uint16_t benchmark_cycles;

    /* Benchmark pin set operation */
    BENCHMARK_START();
    DIO_voidSetPinValue(DIO_PORTA, DIO_PIN0, DIO_PIN_HIGH);
    BENCHMARK_END();

    printf("Pin set time: %lu us\n", CYCLES_TO_US(benchmark_cycles));

    /* Benchmark pin read operation */
    BENCHMARK_START();
    volatile uint8_t pin_state = DIO_u8GetPinValue(DIO_PORTA, DIO_PIN0);
    BENCHMARK_END();

    printf("Pin read time: %lu us\n", CYCLES_TO_US(benchmark_cycles));
}
```

Driver Development Workflow

1. Development Steps

```
1. Requirements Analysis
   ↓
2. Hardware Study (Datasheet)
   ↓
3. Register Mapping
   ↓
4. Interface Design
   ↓
```

```
5. Implementation
   ↓
6. Unit Testing
   ↓
7. Integration Testing
   ↓
8. Documentation
   ↓
9. Code Review
   ↓
10. Release
```

2. Code Review Checklist

- [ ] **Naming Convention**: Functions, variables follow standard
- [ ] **Error Handling**: All edge cases covered
- [ ] **Memory Management**: No leaks, proper initialization
- [ ] **Thread Safety**: ISR-safe operations
- [ ] **Performance**: Optimized critical paths
- [ ] **Documentation**: Complete API documentation
- [ ] **Testing**: Adequate test coverage
- [ ] **Portability**: Hardware-specific code isolated

3. Version Control Strategy

```
Git Branch Structure:
├── main (stable releases)
├── develop (integration branch)
├── feature/driver-name
├── bugfix/issue-description
└── hotfix/critical-fixes

Commit Message Format:
[DRIVER_NAME] Type: Short description

Examples:
[DIO] feat: Add port-level operations
[LCD] fix: Correct 4-bit mode initialization
[ADC] docs: Update API documentation
```

Advanced Driver Features

1. State Machine Implementation

```c
/* Driver State Management */
typedef enum {
    MODULE_STATE_UNINITIALIZED = 0,
    MODULE_STATE_READY,
    MODULE_STATE_BUSY,
    MODULE_STATE_ERROR
} MODULE_State_t;

static MODULE_State_t Module_State = MODULE_STATE_UNINITIALIZED;

MODULE_ErrorStatus_t MODULE_Init(void) {
    if (Module_State != MODULE_STATE_UNINITIALIZED) {
        return MODULE_ERROR;
    }

    /* Initialization code */

    Module_State = MODULE_STATE_READY;
    return MODULE_OK;
}

MODULE_ErrorStatus_t MODULE_StartOperation(void) {
    if (Module_State != MODULE_STATE_READY) {
        return MODULE_ERROR;
    }

    Module_State = MODULE_STATE_BUSY;
    /* Start operation */

    return MODULE_OK;
}
```

2. Timeout Handling

```c
/* Timeout Management */
#define MODULE_TIMEOUT_MS   1000

MODULE_ErrorStatus_t MODULE_WaitWithTimeout(void) {
    uint16_t timeout_counter = 0;

    while ((MODULE_GetStatus() == MODULE_BUSY) &&
```

```
           (timeout_counter < MODULE_TIMEOUT_MS)) {
        _delay_ms(1);
        timeout_counter++;
    }

    if (timeout_counter >= MODULE_TIMEOUT_MS) {
        return MODULE_TIMEOUT;
    }

    return MODULE_OK;
}
```

3. Buffer Management

```
/* Circular Buffer Implementation */
typedef struct {
    uint8_t* buffer;
    uint16_t size;
    uint16_t head;
    uint16_t tail;
    uint16_t count;
} CircularBuffer_t;

boolean BUFFER_IsEmpty(CircularBuffer_t* buffer) {
    return (buffer->count == 0);
}

boolean BUFFER_IsFull(CircularBuffer_t* buffer) {
    return (buffer->count == buffer->size);
}

MODULE_ErrorStatus_t BUFFER_Put(CircularBuffer_t* buffer, uint8_t data)
{
    if (BUFFER_IsFull(buffer)) {
        return MODULE_ERROR;
    }

    buffer->buffer[buffer->head] = data;
    buffer->head = (buffer->head + 1) % buffer->size;
    buffer->count++;

    return MODULE_OK;
}
```

Memory Optimization Techniques

1. Flash Memory Optimization

```
/* Store constant data in program memory */
#include <avr/pgmspace.h>

const char LCD_InitSequence[] PROGMEM = {0x33, 0x32, 0x28, 0x0E, 0x01};
const char ErrorMessages[][20] PROGMEM = {
    "No Error",
    "Invalid Parameter",
    "Timeout Error",
    "Hardware Error"
};

/* Read from program memory */
void LCD_SendInitSequence(void) {
    for (uint8_t i = 0; i < sizeof(LCD_InitSequence); i++) {
        LCD_voidSendCommand(pgm_read_byte(&LCD_InitSequence[i]));
        _delay_ms(5);
    }
}
```

2. RAM Optimization

```
/* Use bit fields for status flags */
typedef struct {
    uint8_t initialized : 1;
    uint8_t busy : 1;
    uint8_t error : 1;
    uint8_t reserved : 5;
} MODULE_Status_t;

/* Use unions for register access */
typedef union {
    uint8_t reg;
    struct {
        uint8_t bit0 : 1;
        uint8_t bit1 : 1;
        uint8_t bit2 : 1;
        uint8_t bit3 : 1;
        uint8_t bit4 : 1;
        uint8_t bit5 : 1;
```

```
        uint8_t bit6 : 1;
        uint8_t bit7 : 1;
    } bits;
} RegisterAccess_t;
```

Power Management Integration

1. Sleep Mode Integration

```
#include <avr/sleep.h>

void MODULE_EnterSleepMode(void) {
    /* Prepare peripherals for sleep */
    ADC_voidDisable();
    UART_voidDisable();

    /* Set sleep mode */
    set_sleep_mode(SLEEP_MODE_PWR_DOWN);
    sleep_enable();

    /* Enable wake-up interrupts */
    sei();

    /* Enter sleep */
    sleep_cpu();

    /* Wake up - disable sleep */
    sleep_disable();

    /* Restore peripherals */
    ADC_voidInit();
    UART_voidInit();
}
```

2. Clock Management

```
/* Dynamic clock scaling */
void SYSTEM_SetClockPrescaler(uint8_t prescaler) {
    cli();  /* Disable interrupts */

    /* Write prescaler to CLKPR */
    CLKPR = (1 << CLKPCE);  /* Enable prescaler change */
    CLKPR = prescaler;      /* Set new prescaler */
```

```
    sei();  /* Enable interrupts */

    /* Update delay functions and timer settings */
    UpdateSystemTimings();
}
```

Documentation Standards

1. API Documentation Template

```
/**
 * @brief Initialize the DIO module
 * @details This function configures the specified pin as input or
output
 *
 * @param Copy_Port Port selection (DIO_PORTA to DIO_PORTD)
 * @param Copy_Pin Pin selection (DIO_PIN0 to DIO_PIN7)
 * @param Copy_Direction Direction (DIO_PIN_INPUT or DIO_PIN_OUTPUT)
 *
 * @return void
 *
 * @pre None
 * @post Pin is configured with specified direction
 *
 * @example
 * ```c
 * DIO_voidSetPinDirection(DIO_PORTA, DIO_PIN0, DIO_PIN_OUTPUT);
 * ```
 *
 * @note This function does not validate input parameters
 * @warning Ensure valid port and pin combinations
 *
 * @author Your Name
 * @date 2025-01-01
 * @version 1.0
 */
void DIO_voidSetPinDirection(DIO_Port_t Copy_Port, DIO_Pin_t Copy_Pin,
DIO_PinDirection_t Copy_Direction);
```

2. Change Log Template

```
# Changelog

## [1.2.0] - 2025-01-15
### Added
- Port-level operations for bulk I/O
- Input pull-up resistor control
- Pin interrupt capability

### Changed
- Improved error handling in all functions
- Optimized register access for better performance

### Fixed
- Pin toggle function timing issue
- Memory leak in configuration structure

### Deprecated
- Old naming convention (will be removed in v2.0)

## [1.1.0] - 2025-01-01
### Added
- Basic pin operations
- Initial release
```

This comprehensive guide provides you with a solid foundation for developing professional-grade ATmega32 drivers. The layered architecture ensures maintainability, reusability, and portability across different projects and microcontrollers.