DM9051 Networking Ethernet Driver Handbook

DM9051 Driver Files

```
dm9051_edriver_r2503_v1.6.1/    # version information
core/
    ├── dm9051.c                # Core functions and operations
    ├── dm9051.h                # Header for core functions
    └── ver.h                   # Version information
hal/
    ├── hal_at32f437.h          # MCU header for AT32F437
    ├── hal_stm32f103.h         # MCU header for STM32F103
    ├── hal_stm32f401.h         # MCU header for STM32F401
    ├── SPI1/
    │   └── at32f437_spi1.c     # SPI1 handling for AT32F437
    ├── SPI2/
    │   ├── stm32f103_spi2.c    # SPI2 handling for STM32F103
    │   └── stm32f401_spi2.c    # SPI2 handling for STM32F401
    └── INT/
        ├── at32f437_int7.c     # Interrupt handling for AT32F437
        ├── stm32f103_int7.c    # Interrupt handling for STM32F103
        └── stm32f401_int7.c    # Interrupt handling for STM32F401
```

DM9051 Networking Application Procedure

Example of uip

POLLING mode

app main program

```
const struct uip_eth_addr domain_addr = {{
    emacETHADDR0,
    emacETHADDR1,
    emacETHADDR2,
    emacETHADDR3,
    emacETHADDR4,
    emacETHADDR5
    }};

struct uip_eth_addr uip_ethaddr = {{
    0, 0, 0, 0, 0, 0
}};
```

```
u16_t uip_len, uip_slen;
u8_t uip_buf[UIP_BUFSIZE];

void vuIP_Conf(void)
{
    dm9051_conf();
}

void vuIP_Init(void)
{
    const uint8_t *raddr = dm9051_init(domain_addr);
    if (raddr)
      memcpy(&uip_ethaddr.addr[0], raddr, MAC_ADDR_LENGTH);
}

int vuIP_Process(void)
{
    uip_len = dm9051_rx(uip_buf);
    ...

    uip_periodic(0);
    if (uip_len > 0)
    {
        uip_arp_out();
        dm9051_tx(uip_buf, uip_len)
    }
    ...
}
```

uip process program

```
#define UIP_ETHTYPE_ARP 0x0806
#define UIP_ETHTYPE_IP  0x0800

struct uip_conn *uip_conn;
struct uip_conn uip_conns[UIP_CONNS];
#define BUF   ((struct arp_hdr *)&uip_buf[0])
#define IPBUF ((struct ethip_hdr *)&uip_buf[0])

void uip_periodic(int conn)
{
    uip_conn = &uip_conns[conn];
    uip_process(UIP_TIMER);
}

void uip_arp_out(void)
{
    u8_t *dest_addr = dest_in_arp_table(IPBUF->destipaddr);
    if (!dest_addr) {
        build_arp_packet(BUF);
        uip_len = sizeof(struct arp_hdr);
        return;
    }

    memcpy(IPBUF->ethhdr.dest.addr, dest_addr, 6);
    memcpy(IPBUF->ethhdr.src.addr, uip_ethaddr.addr, 6);
    IPBUF->ethhdr.type = HTONS(UIP_ETHTYPE_IP);
    uip_len += sizeof(struct uip_eth_hdr);
}
```

INTERRUPT mode

- In interrupt mode, More further operation is required base on above mode program. The essential functions for

  interrupt also provided in the DM9051 driver.

Lwip example

- Lwip example is not provided here, and it is almost the same init and rx/tx procedures and the same poll/interrupt solution

  compare to uip. Also if for the DHCP client function, which does totally operate by the application itself and not aware to

  the driver. This will be mention as well later in this article.

DM9051 Networking Application Considerations

1.IP Address

IP Address Solution:

- User need define a candidated IP address to be used as a prepared setting.

- In DHCP client mode, system will get dynamic IP address from the DHCP server or if fail to

  use this prepared IP address.
- If DHCP client mode is disable, system will directly use the prepared IP address as its static IP address.

Key Considerations:

- Ensure the network interface is up and running before the stack attempts to assign an IP address.
- Handle link status changes to notify the stack of connectivity changes.

2.DHCP Client

The DHCP client functionality is usually part of the TCP/IP stack (like uip, LWIP) and not directly

handled by the driver. However, the driver must support the necessary operations for the stack to

perform DHCP.

Optional Operation Mode:

- DHCP client mode is not mandetory, And only implemented in the application. If not enable DHCP client, system

  will operate with static IP address

Key Considerations:

- Ensure the driver can send and receive packets, as DHCP involves exchanging messages with a DHCP server.
- Handle interrupts and packet reception efficiently to support DHCP message exchanges.

3.Interrupt

Interrupt handling is crucial for efficient network operations. The DM9051 driver provides functions to enable, disable, and handle interrupts.

Optional Operation Mode:

- Interrupt mode is not mandetory, And need implement in both application and driver. If not enable interrupt mode,

  system will operate in polling mode

Relevant Code:

MCU peripheral Code:

```
void EXINT9_5_IRQHandler(void) {
    if (exint_flag_get(EXINT_LINE_7) != RESET) {
      dm9051_interrupt_set(EXINT_LINE_7);
      exint_flag_clear(EXINT_LINE_7);
    }
}
```

DM9051 driver Code:

```
void dm9051_interrupt_set(uint32_t exint_line)
{
    if (innr_inf->ext_line == exint_line){
      dm9051_interrupt_event = 1;
    }
}

int dm9051_interrupt_get(void)
{
  if (!dm9051_driver_cfg->interrupt_mode)
    return 1;
  if (dm9051_interrupt_event) {
    dm9051_interrupt_event = 0;
    cspi_disble_irq();
    return 1;
  }
  return 0;
}

void dm9051_interrupt_reset(void)
{
  if (dm9051_driver_cfg->interrupt_mode) {
    cspi_enable_irq();
  }
}
```

Key Considerations:

- Use `cspi_disable_irq` to disable interrupts by eliminate interrupt mask.
- Use `cspi_enable_irq` to enable interrupts with interrupt masks, and read and clear the ISR register respectively.
- Implement efficient interrupt service routines to handle network events without significant delay.

4.MAC Address

The MAC address is a critical component of network communication, uniquely identifying the device on the network.

MAC Address Solution:

- MAC address usually configured by DM9051 e-fuse (or EEPROM), so only require manual

  operation during product manufacture.
- User can determine his MAC address outside the DM9051 e-fuse (or EEPROM). In this case, user

  can configure his MAC address into the flash memory, read it back from flash memory while system

  startup, and then send as a parameter to dm9051_init(macaddr).

Relevant Code:

```
void cspi_get_par(uint8_t *adr)
{
  cspi_read_regs(DM9051_PAR, adr, 6, CS_EACH);
}

void cspi_set_par(const uint8_t *adr)
{
  cspi_write_regs(DM9051_PAR, adr, 6);
}
```

```
void cspi_macaddr_process(const uint8_t *param_adr)
{
  uint8_t c_adr[6];

  if (param_adr && validate_macaddr(param_adr)) {
    identify_eth_mac(param_adr);
    cspi_set_par(identified_eth_mac());
    return;
  }

  if (validate_macaddr(c_adr)) {
    identify_eth_mac(c_adr);
  }
  else
  {
    /* while invalid, do configure MAC address */
    identify_eth_mac(NULL);
    cspi_set_par(identified_eth_mac());
  }
}
```

Key Considerations:

- Use `cspi_set_par` to configure the MAC address registers with a valid MAC address.
- Validate MAC addresses using `validate_macaddr` to ensure they are not multicast or zero addresses.
- Handle MAC address initialization in `init_macaddr_process`, which sets the MAC address based on input or defaults to a random address if necessary.

DM9051 Networking Driver Documentation

1. Driver Model Settings

```
struct instance_data {
    // Per-device configurations
    struct driver_config {
        int interrupt_mode;      // MODE_POLL/MODE_INTERRUPT
        // Mode-specific configuration
        struct mod_config {
            int skb_wb_mode;     // Write-back buffer mode
            int tx_mode;         // FORCE_TX_CONTI_ON/OFF
            int checksuming;     // Checksum offload enable
        } mode;
    } dev;
    int force_mode_10m;
    int force_flow_control;
    int force_rcr_all;
    int force_stop_if_dm9051_not_found;
};
```

2. Eaxample Settings

```
#define INTR_POLL_SEL MODE_POLL

struct instance_data dm9051_def_inf = {
    {
        INTR_POLL_SEL,                // Default to polling mode
        {
         SKB_WB_ON,                   // Write-back enabled
         FORCE_TX_CONTI_OFF,          // Standard TX mode
         DEFAULT_CHECKSUM_OFF,        // Checksum offload disabled
        },
    }
    FORCE_MODE_10M_OFF,               // Auto-speed detection
    FORCE_FLOW_CONTROL_ON,            // Flow control enabled
    FORCE_RCR_ALL_ON,                 // Multicast filtering
    FORCE_STOP_IF_DM9051_NOT_FOUND_OFF// Not found stop
};
```

3. APIs

Here are the key functions:

1. dm9051_conf (SPI/Interrupt Initialization):

   ```
   int dm9051_conf(void)
   {
       HAL_boards_initialize_spi();
       if (dm9051_driver_cfg->interrupt_mode)
           dm9051_driver_cfg->ext_line = HAL_boards_initialize_int();
       return dm9051_driver_cfg->interrupt_mode;
   }
   ```

Initializes SPI interface

Configures interrupts if in interrupt mode

Sets up hardware abstraction layer

2. dm9051_init (Core Initialization):

```
const uint8_t *dm9051_init(const uint8_t *adr)
{
    /* verify device */
    if (!env_chip_id_and_ticks())
    {
        check_force_stop();
        printf("\r\n");
        return NULL;
    }

    return init_setup(adr);
}
```

Performs chip ID verification

Configures MAC address (init_setup(adr))

Sets up receive parameters and interrupts (init_setup(adr))

Handles reset sequence and flow control (init_setup(adr))

3. dm9051_rx (Packet Reception):

```
uint16_t dm9051_rx(uint8_t *buff)
{
    if (cspi_rx_ready())
        return cspi_rx_read(buff, cspi_rx_head());
    return 0;
}
```

Checks RX ready status

Reads packet header to get length

Retrieves packet data with bus security XOR

Handles padding for 16-bit alignment

4. dm9051_tx (Packet Transmission):

```
void dm9051_tx(uint8_t *buf, uint16_t len)
{
    if (dm9051_driver_cfg->mode.tx_mode == FORCE_TX_CONTI_ON)
        dm9051_tx_conti(buf, len);
    else
        dm9051_tx_req(buf, len);
}
```

Supports two transmission modes:

Continuous mode with pre-checked buffer space

Standard mode with ready-state polling

Handles packet padding

Implements timeout mechanisms for reliable delivery

5. Key characteristics across all functions:

- Heavy use of hardware abstraction layer (HAL) for SPI operations
- Support for both polling and interrupt modes
- Error handling through status register checks
- Configuration-driven operation through instance_data structure

4. Porting Guide

Customization: Porting to user's MCU device

How to create new hardware support files, refer to the implementation of STM32F103:

- The spi and interrupt Parameter : SPI3 and EXINT_LINE_0

1. New Hardware File Structure

```
hal/
├── hal_newmcu.h # MCU header for new mcu
├── SPI3/
│  └── newmcu_spi3.c # SPI3 handling for new mcu
└── INT/
└── newmcu_int0.c # Interrupt handling for new mcu
```

2. Key Components to Implement

2.1 hal_newmcu.h (MCU definition header)

```
#include "stdio.h"
#include "newmcu.h"
```

2.2 newmcu_spi3.c (SPI Implementation)

1. SPI Configuration

void hal_spi_initialize(void) {
// Enable peripheral clocks
RCC->APB2ENR |= RCC_APB2ENR_IOPAEN | RCC_APB2ENR_AFIOEN;
RCC->APB1ENR |= RCC_APB1ENR_SPI2EN;

```
    // Configure GPIO pins (SCK/MOSI/MISO)
    GPIO_InitTypeDef GPIO_InitStruct = {
        .GPIO_Pin = GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15,
        .GPIO_Mode = GPIO_Mode_AF_PP,
        .GPIO_Speed = GPIO_Speed_50MHz
    };
    GPIO_Init(GPIOB, &GPIO_InitStruct);

    // Configure SPI peripheral
    SPI_InitTypeDef SPI_InitStruct = {
        .SPI_Direction = SPI_Direction_2Lines_FullDuplex,
        .SPI_Mode = SPI_Mode_Master,
        .SPI_DataSize = SPI_DataSize_8b,
        .SPI_CPOL = SPI_CPOL_Low,
        .SPI_CPHA = SPI_CPHA_1Edge,
        .SPI_NSS = SPI_NSS_Soft,
        .SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_4,
        .SPI_FirstBit = SPI_FirstBit_MSB
    };
    SPI_Init(SPI2, &SPI_InitStruct);
    SPI_Cmd(SPI2, ENABLE);
}

2. SPI Transaction Functions
static uint8_t SPI_DM9051_SendByte(uint8_t byte) {
    while (SPI_I2S_GetFlagStatus(SPI2, SPI_I2S_FLAG_TXE) == 0);
    SPI_I2S_SendData(SPI2, byte);
    while (SPI_I2S_GetFlagStatus(SPI2, SPI_I2S_FLAG_RXNE) == 0);
    return SPI_I2S_ReceiveData(SPI2);
}

3. Chip Select Control
static void hal_stdpin_lo(void) { /* CS Low */ }
static void hal_stdpin_hi(void) { /* CS High */ }
```

```
4. HAL - SPI Data Read/Write Function
static void hal_spi_data_read(uint8_t reg, uint8_t *pd) {
    SPI_DM9051_SendByte(reg);
    *pd = SPI_DM9051_SendByte(0x0);
}
static void hal_spi_data_write(uint8_t reg, uint8_t val) {
    SPI_DM9051_SendByte(reg);
    SPI_DM9051_SendByte(val);
}

// HAL - Memory Read and Write Functions
static void hal_spi_mem_read(uint8_t reg, uint8_t *buf, uint16_t len) {
    uint16_t i;
    SPI_DM9051_SendByte(reg);
    for (i = 0; i < len; i++)
    buf[i] = SPI_DM9051_SendByte(0x0);
}
static void hal_spi_mem_write(uint8_t reg, uint8_t *buf, uint16_t len) {
    uint16_t i;
    SPI_DM9051_SendByte(reg);
    for (i = 0; i < len; i++)
    SPI_DM9051_SendByte(buf[i]);
}

5. SPI register read/write
uint8_t hal_read_reg(uint8_t reg) {
    uint8_t val;
    hal_stdpin_lo();
    hal_spi_data_read(reg | OPC_REG_R, &val);
    hal_stdpin_hi();
    return val;
}
void hal_write_reg(uint8_t reg, uint8_t val) {
    hal_stdpin_lo();
    hal_spi_data_write(reg | OPC_REG_W, val);
    hal_stdpin_hi();
}

5. SPI memory read/write
void hal_read_mem(uint8_t *buf, uint16_t len) {
    hal_stdpin_lo();
    hal_spi_mem_read(DM9051_MRCMD | OPC_REG_R, buf, len);
    hal_stdpin_hi();
}
void hal_write_mem(uint8_t *buf, uint16_t len) {
    hal_stdpin_lo();
    hal_spi_mem_write(DM9051_MWCMD | OPC_REG_W, buf, len);
    hal_stdpin_hi();
}
```

2.3 newmcu_int0.c (Interrupt Implementation):

1. Interrupt Configuration

uint32_t hal_int_initialize(void) {
// GPIO Configuration
GPIO_InitTypeDef GPIO_InitStruct = {
.GPIO_Pin = GPIO_Pin_7,
.GPIO_Mode = GPIO_Mode_IPU,
.GPIO_Speed = GPIO_Speed_10MHz
};
GPIO_Init(GPIOC, &GPIO_InitStruct);

```
    // EXTI Configuration
    EXTI_InitTypeDef EXTI_InitStruct = {
        .EXTI_Line = EXTI_Line7,
        .EXTI_Mode = EXTI_Mode_Interrupt,
        .EXTI_Trigger = EXTI_Trigger_Falling,
        .EXTI_LineCmd = ENABLE
    };
    EXTI_Init(&EXTI_InitStruct);

    // NVIC Configuration
    NVIC_InitTypeDef NVIC_InitStruct = {
        .NVIC_IRQChannel = EXTI9_5_IRQn,
        .NVIC_IRQChannelPreemptionPriority = 1,
        .NVIC_IRQChannelSubPriority = 0,
        .NVIC_IRQChannelCmd = ENABLE
    };
    NVIC_Init(&NVIC_InitStruct);

    return EXTI_Line7;
}

2. Interrupt Control
void hal_enable_mcu_irq(void) {
    NVIC_EnableIRQ(EXTI9_5_IRQn);
}

void hal_disable_mcu_irq(void) {
    NVIC_DisableIRQ(EXTI9_5_IRQn);
}

3. Interrupt EXTI line
uint32_t hal_irqline(void){
    return EXTI_Line7;
}
```

3. Implementation Requirements

```
1. SPI Requirements:

Implement full-duplex SPI communication
Support 8-bit data transfers
Handle clock polarity/phase (CPOL/CPHA)
Manage chip select (CS) line
Support multiple SPI speeds (via prescaler)

2. Interrupt Requirements:

Configure GPIO as interrupt source
Set falling/rising edge detection
Implement NVIC priority configuration
Provide enable/disable control
Handle interrupt pending flags
```

4. Hardware Abstraction Layer (HAL) Interface

```
/* Required Interface Functions */

// SPI
void hal_spi_initialize(void);
uint8_t hal_read_reg(uint8_t reg);
void hal_write_reg(uint8_t reg, uint8_t val);
void hal_read_mem(uint8_t *buf, uint16_t len);
void hal_write_mem(uint8_t *buf, uint16_t len);

// Interrupt
void hal_int_initialize(void);
void hal_enable_mcu_irq(void);
void hal_disable_mcu_irq(void);
uint32_t hal_irqline(void);
```

5. Note

- This structure maintains compatibility with the existing DM9051 driver
  while allowing customization
  for new hardware targets.