问题描述

在RT1052 SDK软件库中的Lwip例程的PHY驱动是基于KSZ8081RNB，但客户在设计评估板时，选用的是 LAN8720芯片，如下图所示，

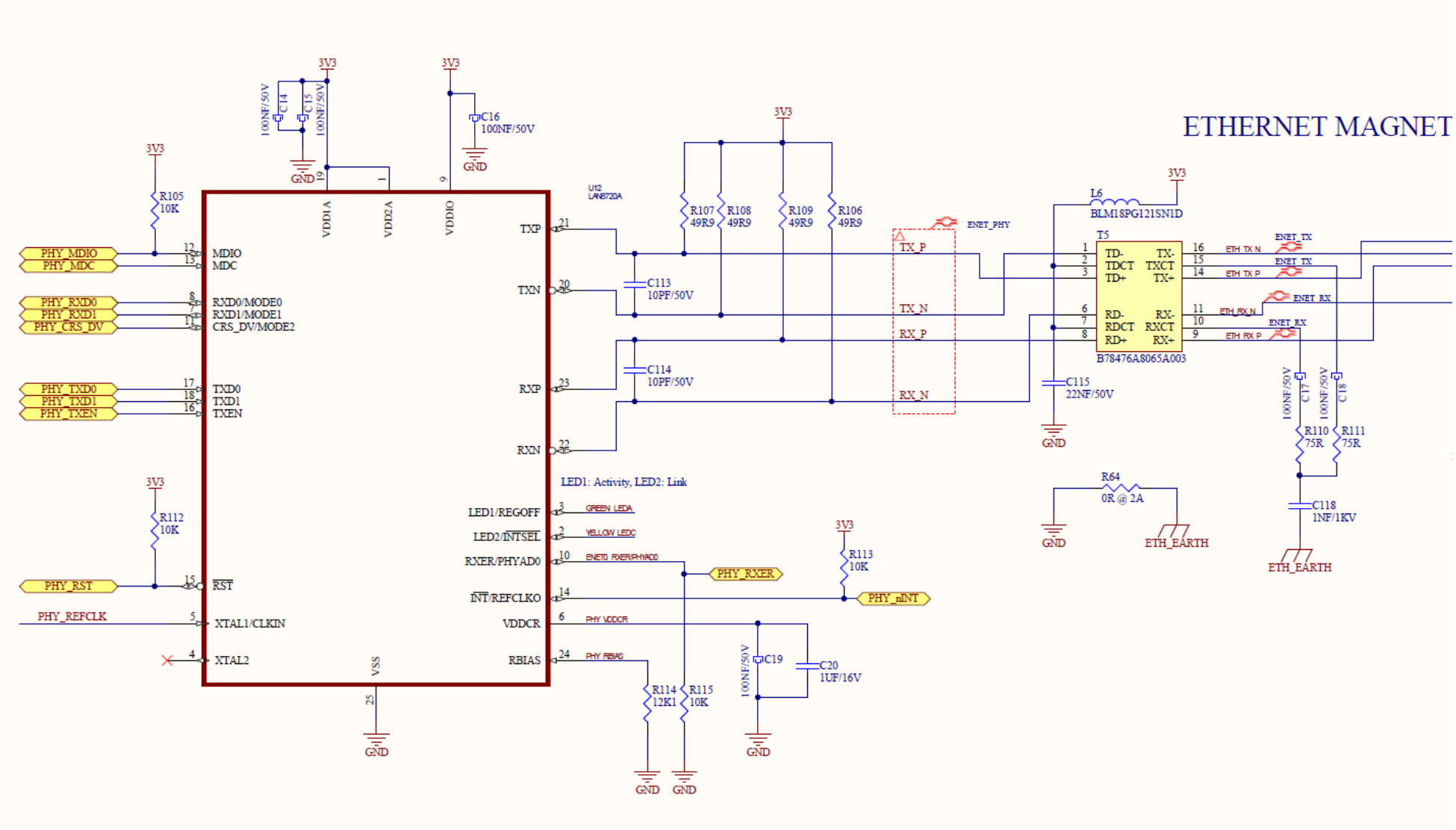


图1 LAN8720电路图

这就需要调整SDK软件库中的硬件驱动以适配LAN8720芯片, 话不多说开整。

代码修改

以lwip_ping_bm工程代码为例。

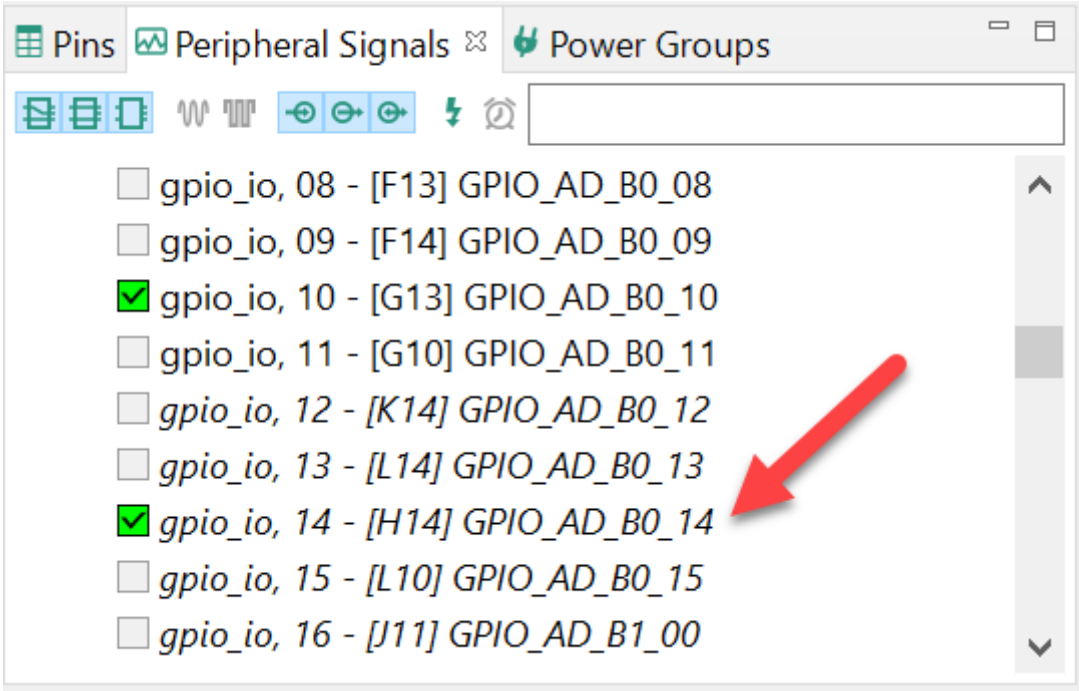
1) 引脚

对比MIMXRT1050 EVK板的PHY芯片连接电路，发现客户评估板基本上沿袭了MIMXRT1050 EVK板的电路设计，除了ENET_RST连接脚不同外，RMII接口和ENET_INT引脚分布都是一样的。

接口引脚	MIMXRT1050 EVK	客户评估板
RMII_MDC	GPIO_EMC_40	GPIO_EMC_40
RMII_MDIO	GPIO_EMC_41	GPIO_EMC_41
RMII_CRS_DV	GPIO_B1_06	GPIO_B1_06
RMII_RXD[0]	GPIO_B1_04	GPIO_B1_04
RMII_RXD[1]	GPIO_B1_05	GPIO_B1_05
RMII_RXER	GPIO_B1_11	GPIO_B1_11
RMII_TXD[0]	GPIO_B1_07	GPIO_B1_07
RMII_TXD[1]	GPIO_B1_08	GPIO_B1_08

接口引脚	MIMXRT1050 EVK	客户评估板
RMII_REF_CLK	GPIO_B1_10	GPIO_B1_10
ENET_INT	GPIO_AD_B0_10	GPIO_AD_B0_10
PHY_RST	**GPIO_AD_B0_09**	**GPIO_AD_B0_14**

首先，通过MCUXpresso IDE集成的config tools修改连接ENET_RST的GPIO引脚（如下图所示）并生成对应GPIO配置代码，



跟着修改main() 函数中复位LAN8720芯片的代码（如下所示）

```
/* pull up the ENET_INT before RESET. */
GPIO_WritePinOutput(GPIO1, 10, 1);
//GPIO_WritePinOutput(GPIO1, 9, 0);
GPIO_WritePinOutput(GPIO1, 14, 0);
```

```
    delay();
    //GPIO_WritePinOutput(GPIO1, 9, 1);
    GPIO_WritePinOutput(GPIO1, 14, 1);
```

2) 时钟源

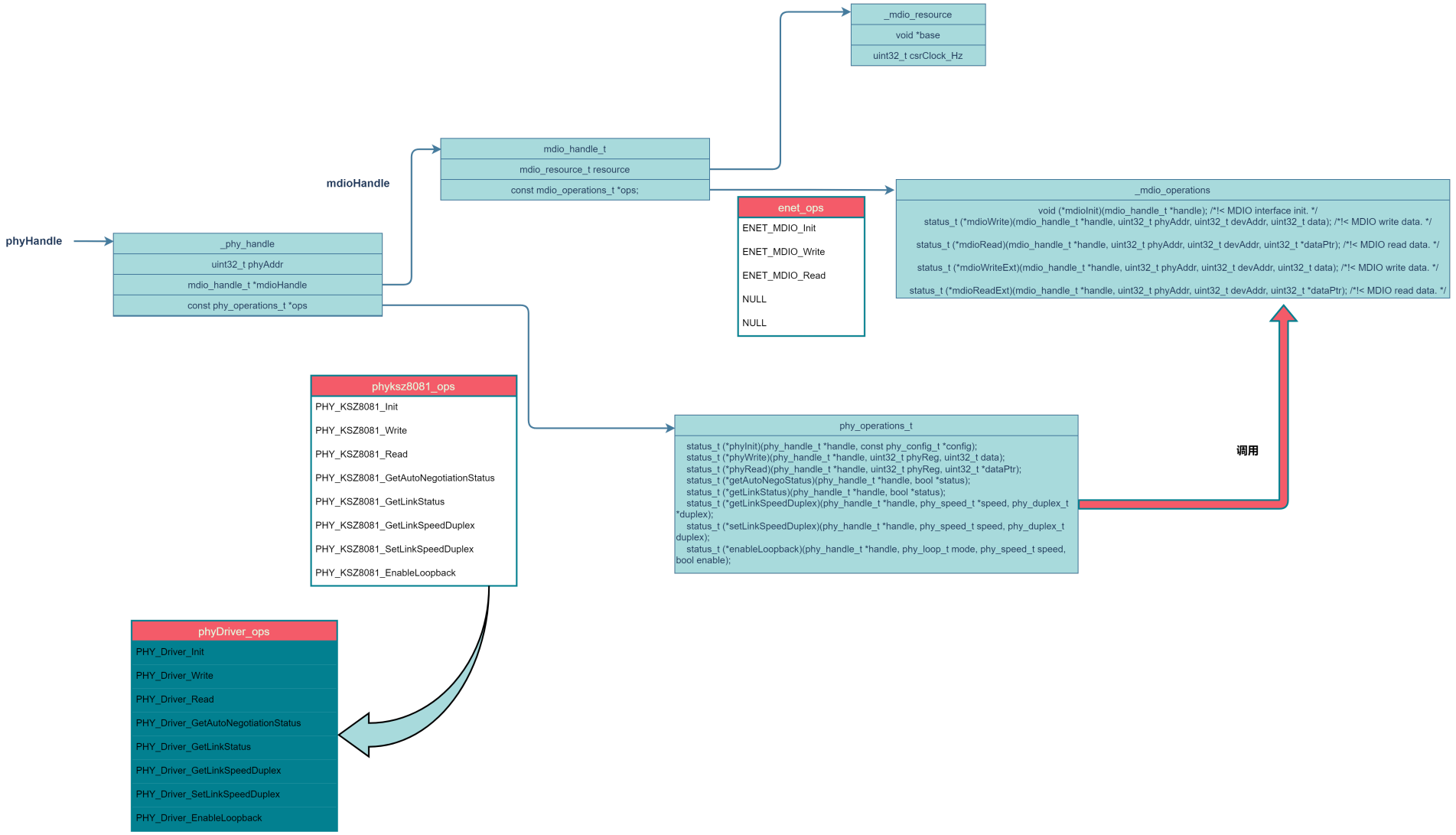
关于RT1050如何产生RMII_REF_CLK时钟源，请参考以前的文章**《RT1050 ENET_REF引脚解惑》**，由于MIMXRT1050 EVK板和客户评估板的RMII_REF_CLK引脚是相同的，故时钟源产生代码无需修改。

3) LAN8720芯片驱动

代码修改集中在fsl_phyksz8081.c文件，对比LAN8720和KSZ8081RNB芯片的数据手册后，确定修改点如下：

- `#define PHY_CONTROL1_REG 0x1FU//0x1EU /*!< The PHY control one register. */`
- `#define PHY_CONTROL_ID1 0x07 //0x22U /*!< The PHY ID1 */`
- `#define PHY_CTL1_SPEEDUPLX_MASK 0x001CU //0x0007U /*!< The PHY speed and duplex mask. */`

下图是lwip_ping_bm的PHY driver调用关系图，



我们可以改造phyksz8081_ops结构体中的函数指针项对应的代码，然后通过宏定义选择使其适用KSZ8081RNB和LAN8720芯片，具体代码修改过程如下：

1. 重命名fsl_phyksz8081.c和fsl_phyksz8081.h为fsl_phy_driver.c和fsl_phy_driver.h
2. 修改fsl_phy_driver.c中的函数代码

```
#include "fsl_phy_driver.h"

/*******************************************************************************
 * Definitions
 ******************************************************************************/

/*! @brief Defines the PHY Driver vendor defined registers. */

#if defined(PHY_KSZ8081)
#define PHY_CONTROL1_REG 0x1EU          /*!< The PHY control one register. */
#define PHY_CONTROL2_REG 0x1FU          /*!< The PHY control two register. */
#define PHY_CONTROL_ID1 0x22U /*!< The PHY ID1*/
#endif

#if defined(PHY_LAN8720A)
#define PHY_MODE_CTRLSTATUS_REG         0x11
/*!< The PHY Mode Control/Status Register. */
#define PHY_INTSRCFLAG_REG              0x1D            /*!< The PHY Interrupt Source Flag register. */
#define PHY_INTMASK_REG                 0x1EU           /*!< The PHY control Interrupt Mask register. */
#define PHY_CTRLSTATUS_REG              0x1FU           /*!< The PHY control Status  register. */
#define PHY_CONTROL_ID1                 0x07U           /*!< The PHY ID1*/

#define PHY_INTMASK_REG_MASK            0xFE
#define PHY_MODE_CTRLSTATUS_REMOTELOOP_MASK              0x200U
#endif


/*! @brief Defines the mask flag of operation mode in control registers */
#define PHY_CTL2_REMOTELOOP_MASK    0x0004U /*!< The PHY remote loopback mask. */
```

```
#define PHY_CTL2_REFCLK_SELECT_MASK 0x0080U /*!< The PHY RMII reference clock select. */
#define PHY_CTL1_10HALFDUPLEX_MASK  0x0001U /*!< The PHY 10M half duplex mask. */
#define PHY_CTL1_100HALFDUPLEX_MASK 0x0002U /*!< The PHY 100M half duplex mask. */
#define PHY_CTL1_10FULLDUPLEX_MASK  0x0005U /*!< The PHY 10M full duplex mask. */
#define PHY_CTL1_100FULLDUPLEX_MASK 0x0006U /*!< The PHY 100M full duplex mask. */
#define PHY_CTL1_SPEEDUPLX_MASK     0x001CU //0x0007U /*!< The PHY speed and duplex mask. */
#define PHY_CTL1_ENERGYDETECT_MASK  0x10U   /*!< The PHY signal present on rx differential pair. */
#define PHY_CTL1_LINKUP_MASK        0x100U  /*!< The PHY link up. */
#define PHY_LINK_READY_MASK         (PHY_CTL1_ENERGYDETECT_MASK | PHY_CTL1_LINKUP_MASK)

/*! @brief Defines the timeout macro. */
#define PHY_READID_TIMEOUT_COUNT 1000U
/*! @brief Defines the timeout macro. */
#define PHY_TIMEOUT_COUNT 0x3FFFFFU

/*******************************************************************************
 * Prototypes
 ******************************************************************************/

/*******************************************************************************
 * Variables
 ******************************************************************************/
const phy_operations_t phyDriver_ops = { .phyInit            = PHY_Driver_Init,
                                         .phyWrite           = PHY_Driver_Write,
                                         .phyRead            = PHY_Driver_Read,
                                         .getAutoNegoStatus  = PHY_Driver_GetAutoNegotiationStatus,
                                         .getLinkStatus      = PHY_Driver_GetLinkStatus,
                                         .getLinkSpeedDuplex = PHY_Driver_GetLinkSpeedDuplex,
                                         .setLinkSpeedDuplex = PHY_Driver_SetLinkSpeedDuplex,
                                         .enableLoopback     = PHY_Driver_EnableLoopback};

/*******************************************************************************
 * Code
```

```
 *****************************************************************************/

status_t PHY_Driver_Init(phy_handle_t *handle, const phy_config_t *config)
{
 uint32_t bssReg;

 uint32_t counter  = PHY_READID_TIMEOUT_COUNT;
    status_t result   = kStatus_Success;
    uint32_t regValue = 0;

    uint32_t timeDelay;
    /* Init MDIO interface. */
    MDIO_Init(handle->mdioHandle);

    /* Assign phy address. */
    handle->phyAddr = config->phyAddr;

    /* Check PHY ID. */
    do
    {
        result = MDIO_Read(handle->mdioHandle, handle->phyAddr, PHY_ID1_REG, &regValue);
        if (result != kStatus_Success)
        {
            return result;
        }
        counter--;
    } while ((regValue != PHY_CONTROL_ID1) && (counter != 0U));

    if (counter == 0U)
    {
        return kStatus_Fail;
    }

    /* Reset PHY. */
    result = MDIO_Write(handle->mdioHandle, handle->phyAddr, PHY_BASICCONTROL_REG, PHY_BCTL_RESET_MASK);
```

```
    if (result == kStatus_Success)
    {
/*#if defined(FSL_FEATURE_PHYDriver_USE_RMII50M_MODE)
        result = MDIO_Read(handle->mdioHandle, handle->phyAddr, PHY_CONTROL2_REG, &regValue);
        if (result != kStatus_Success)
        {
            return result;
        }
        result =
            MDIO_Write(handle->mdioHandle, handle->phyAddr, PHY_CONTROL2_REG, (regValue | PHY_CTL2_REFCLK_SELECT_MASK));
        if (result != kStatus_Success)
        {
            return result;
        }
#endif  FSL_FEATURE_PHYDriver_USE_RMII50M_MODE */
#if defined(PHY_LAN8720A)

          /* Enable Interrupt */

    result = MDIO_Read(handle->mdioHandle, handle->phyAddr, PHY_INTMASK_REG, &regValue);

    if (result == kStatus_Success)
        {
            MDIO_Write(handle->mdioHandle, handle->phyAddr,PHY_INTMASK_REG, (regValue | PHY_INTMASK_REG_MASK));
        }

#endif


        if (config->autoNeg)
        {
            /* Set the auto-negotiation then start it. */
            result =
                MDIO_Write(handle->mdioHandle, handle->phyAddr, PHY_AUTONEG_ADVERTISE_REG,
```

```
                    (PHY_100BASETX_FULLDUPLEX_MASK | PHY_100BASETX_HALFDUPLEX_MASK |
                     PHY_10BASETX_FULLDUPLEX_MASK | PHY_10BASETX_HALFDUPLEX_MASK | PHY_IEEE802_3_SELECTOR_MASK));
        if (result == kStatus_Success)
        {
            result = MDIO_Write(handle->mdioHandle, handle->phyAddr, PHY_BASICCONTROL_REG,
                                (PHY_BCTL_AUTONEG_MASK | PHY_BCTL_RESTART_AUTONEG_MASK));
            if (result == kStatus_Success)
            {
                /* Check auto negotiation complete. */
                while (counter --)
                {
                    result = MDIO_Read(handle->mdioHandle, handle->phyAddr, PHY_BASICSTATUS_REG, &bssReg);
                    if ( result == kStatus_Success)
                    {
                        if ((bssReg & PHY_BSTATUS_AUTONEGCOMP_MASK) != 0)
                        {
                            /* Wait a moment for Phy status stable. */
                            for (timeDelay = 0; timeDelay < PHY_TIMEOUT_COUNT; timeDelay ++)
                            {
                                __ASM("nop");
                            }
                            break;
                        }
                    }

                    if (!counter)
                    {
                        return kStatus_PHY_AutoNegotiateFail;
                    }
                }
            }
        }
    }
```

```
        else
        {
            /* This PHY only supports 10/100M speed. */
            assert(config->speed <= kPHY_Speed100M);

            /* Disable isolate mode */
            result = MDIO_Read(handle->mdioHandle, handle->phyAddr, PHY_BASICCONTROL_REG, &regValue);
            if (result != kStatus_Success)
            {
                return result;
            }
            regValue &= ~PHY_BCTL_ISOLATE_MASK;
            result = MDIO_Write(handle->mdioHandle, handle->phyAddr, PHY_BASICCONTROL_REG, regValue);
            if (result != kStatus_Success)
            {
                return result;
            }

            /* Disable the auto-negotiation and set user-defined speed/duplex configuration. */
            result = PHY_Driver_SetLinkSpeedDuplex(handle, config->speed, config->duplex);
        }
    }
    return result;
}

status_t PHY_Driver_Write(phy_handle_t *handle, uint32_t phyReg, uint32_t data)
{
    return MDIO_Write(handle->mdioHandle, handle->phyAddr, phyReg, data);
}

status_t PHY_Driver_Read(phy_handle_t *handle, uint32_t phyReg, uint32_t *dataPtr)
{
    return MDIO_Read(handle->mdioHandle, handle->phyAddr, phyReg, dataPtr);
}
```

```
status_t PHY_Driver_GetAutoNegotiationStatus(phy_handle_t *handle, bool *status)
{
    assert(status);

    status_t result;
    uint32_t regValue;

    *status = false;

    /* Check auto negotiation complete. */
    result = MDIO_Read(handle->mdioHandle, handle->phyAddr, PHY_BASICSTATUS_REG, &regValue);
    if (result == kStatus_Success)
    {
        if ((regValue & PHY_BSTATUS_AUTONEGCOMP_MASK) != 0U)
        {
            *status = true;
        }
    }
    return result;
}

status_t PHY_Driver_GetLinkStatus(phy_handle_t *handle, bool *status)
{
    assert(status);

    status_t result;
    uint32_t regValue;

    /* Read the basic status register. */
    result = MDIO_Read(handle->mdioHandle, handle->phyAddr, PHY_BASICSTATUS_REG, &regValue);
    if (result == kStatus_Success)
    {
        if ((PHY_BSTATUS_LINKSTATUS_MASK & regValue) != 0U)
```

```
        {
            /* Link up. */
            *status = true;
        }
        else
        {
            /* Link down. */
            *status = false;
        }
    }
    return result;
}

status_t PHY_Driver_GetLinkSpeedDuplex(phy_handle_t *handle, phy_speed_t *speed, phy_duplex_t *duplex)
{
    assert(!((speed == NULL) && (duplex == NULL)));

    status_t result;
    uint32_t regValue;
    uint32_t flag;

    /* Read the control register. */
#if defined(PHY_KSZ8081)
    result = MDIO_Read(handle->mdioHandle, handle->phyAddr, PHY_CONTROL1_REG, &regValue);
    if (result == kStatus_Success)
    {
        if (speed != NULL)
        {
            flag = regValue & PHY_CTL1_SPEEDUPLX_MASK;
            if ((PHY_CTL1_100HALFDUPLEX_MASK == flag) || (PHY_CTL1_100FULLDUPLEX_MASK == flag))
            {
                *speed = kPHY_Speed100M;
            }
            else
```

```
            {
                *speed = kPHY_Speed10M;
            }
        }
#endif
#if defined(PHY_LAN8720A)
    result = MDIO_Read(handle->mdioHandle, handle->phyAddr, PHY_CTRLSTATUS_REG, &regValue);
    if (result == kStatus_Success)
    {
        if (speed != NULL)
        {
            flag = regValue & PHY_CTL1_SPEEDUPLX_MASK;
            if ((PHY_CTL1_100HALFDUPLEX_MASK == flag) || (PHY_CTL1_100FULLDUPLEX_MASK == flag))
            {
                *speed = kPHY_Speed100M;
            }
            else
            {
                *speed = kPHY_Speed10M;
            }
        }
#endif
        if (duplex != NULL)
        {
            flag = regValue & PHY_CTL1_SPEEDUPLX_MASK;
            if ((PHY_CTL1_10FULLDUPLEX_MASK == flag) || (PHY_CTL1_100FULLDUPLEX_MASK == flag))
            {
                *duplex = kPHY_FullDuplex;
            }
            else
            {
                *duplex = kPHY_HalfDuplex;
            }
```

```
        }
    }
    return result;
}

status_t PHY_Driver_SetLinkSpeedDuplex(phy_handle_t *handle, phy_speed_t speed, phy_duplex_t duplex)
{
    /* This PHY only supports 10/100M speed. */
    assert(speed <= kPHY_Speed100M);

    status_t result;
    uint32_t regValue;

    result = MDIO_Read(handle->mdioHandle, handle->phyAddr, PHY_BASICCONTROL_REG, &regValue);
    if (result == kStatus_Success)
    {
        /* Disable the auto-negotiation and set according to user-defined configuration. */
        regValue &= ~PHY_BCTL_AUTONEG_MASK;
        if (speed == kPHY_Speed100M)
        {
            regValue |= PHY_BCTL_SPEED0_MASK;
        }
        else
        {
            regValue &= ~PHY_BCTL_SPEED0_MASK;
        }
        if (duplex == kPHY_FullDuplex)
        {
            regValue |= PHY_BCTL_DUPLEX_MASK;
        }
        else
        {
            regValue &= ~PHY_BCTL_DUPLEX_MASK;
        }
```

```
        result = MDIO_Write(handle->mdioHandle, handle->phyAddr, PHY_BASICCONTROL_REG, regValue);
    }
    return result;
}

status_t PHY_Driver_EnableLoopback(phy_handle_t *handle, phy_loop_t mode, phy_speed_t speed, bool enable)
{
    /* This PHY only supports local/remote loopback and 10/100M speed. */
    assert(mode <= kPHY_RemoteLoop);
    assert(speed <= kPHY_Speed100M);

    status_t result;
    uint32_t regValue;

    /* Set the loop mode. */
    if (enable)
    {
        if (mode == kPHY_LocalLoop)
        {
            if (speed == kPHY_Speed100M)
            {
                regValue = PHY_BCTL_SPEED0_MASK | PHY_BCTL_DUPLEX_MASK | PHY_BCTL_LOOP_MASK;
            }
            else
            {
                regValue = PHY_BCTL_DUPLEX_MASK | PHY_BCTL_LOOP_MASK;
            }
            return MDIO_Write(handle->mdioHandle, handle->phyAddr, PHY_BASICCONTROL_REG, regValue);
        }
        else
        {
            /* Remote loopback only supports 100M full-duplex. */
/*            assert(speed == kPHY_Speed100M);
```

```
            regValue = PHY_BCTL_SPEED0_MASK | PHY_BCTL_DUPLEX_MASK | PHY_BCTL_LOOP_MASK;
            result   = MDIO_Write(handle->mdioHandle, handle->phyAddr, PHY_BASICCONTROL_REG, regValue);
            if (result != kStatus_Success)
            {
                return result;
            }*/
            /* Set the remote loopback bit. */

        /* First read the current status in control register. */
#if defined(PHY_KSZ8081)
            result = MDIO_Read(handle->mdioHandle, handle->phyAddr, PHY_CONTROL2_REG, &regValue);
            if (result == kStatus_Success)
            {
                return MDIO_Write(handle->mdioHandle, handle->phyAddr, PHY_CONTROL2_REG,\
                                  (regValue | PHY_CTL2_REMOTELOOP_MASK));
            }
#endif
#if defined(PHY_LAN8720A)

            result = MDIO_Read(handle->mdioHandle, handle->phyAddr, PHY_MODE_CTRLSTATUS_REG, &regValue);

            if (result == kStatus_Success)

            {

                return MDIO_Write(handle->mdioHandle, handle->phyAddr, PHY_MODE_CTRLSTATUS_REG, \
                (regValue | PHY_MODE_CTRLSTATUS_REMOTELOOP_MASK));
            }
#endif
        }
    }
    else
    {
```

```
        /* Disable the loop mode. */
        if (mode == kPHY_LocalLoop)
        {
            /* First read the current status in control register. */
            result = MDIO_Read(handle->mdioHandle, handle->phyAddr, PHY_BASICCONTROL_REG, &regValue);
            if (result == kStatus_Success)
            {
                regValue &= ~PHY_BCTL_LOOP_MASK;
                return MDIO_Write(handle->mdioHandle, handle->phyAddr, PHY_BASICCONTROL_REG,
                                  (regValue | PHY_BCTL_RESTART_AUTONEG_MASK));
            }
        }
        else
        {
#if defined(PHY_KSZ8081)
            /* First read the current status in control one register. */
            result = MDIO_Read(handle->mdioHandle, handle->phyAddr, PHY_CONTROL2_REG, &regValue);
            if (result == kStatus_Success)
            {
                return MDIO_Write(handle->mdioHandle, handle->phyAddr, PHY_CONTROL2_REG,
                                  (regValue & ~PHY_CTL2_REMOTELOOP_MASK));
            }
#endif
#if defined(PHY_LAN8720A)

            result = MDIO_Read(handle->mdioHandle, handle->phyAddr, PHY_MODE_CTRLSTATUS_REG, &regValue);

            if (result == kStatus_Success)
            {
                return MDIO_Write(handle->mdioHandle, handle->phyAddr, PHY_MODE_CTRLSTATUS_REG, (regValue & ~PHY_MODE_CTRLSTATUS_REMOTELOOP_MASK)
            }
#endif
```

```
        }
    }
    return result;
}
```

3. 修改fsl_phy_driver.h中的函数声明，并将phyksz8081_ops结构体名称重命名为 `phyDriver_ops`

```
/*! @brief PHY driver version */
#define FSL_PHY_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

/*! @brief PHY operations structure. */
extern const phy_operations_t phyDriver_ops;

/*******************************************************************************
 * API
 ******************************************************************************/

#if defined(__cplusplus)
extern "C" {
#endif

/*!
 * @name PHY Driver
 * @{
 */

/*!
 * @brief Initializes PHY.
 *
 *  This function initialize PHY.
 *
 * @param handle       PHY device handle.
```

```
 * @param config       Pointer to structure of phy_config_t.
 * @retval kStatus_Success  PHY initialization succeeds
 * @retval kStatus_Fail  PHY initialization fails
 * @retval kStatus_PHY_SMIVisitTimeout  PHY SMI visit time out
 */
status_t PHY_Driver_Init(phy_handle_t *handle, const phy_config_t *config);

/*!
 * @brief PHY Write function. This function writes data over the SMI to
 * the specified PHY register. This function is called by all PHY interfaces.
 *
 * @param handle  PHY device handle.
 * @param phyReg  The PHY register.
 * @param data    The data written to the PHY register.
 * @retval kStatus_Success     PHY write success
 * @retval kStatus_PHY_SMIVisitTimeout  PHY SMI visit time out
 */
status_t PHY_Driver_Write(phy_handle_t *handle, uint32_t phyReg, uint32_t data);

/*!
 * @brief PHY Read function. This interface read data over the SMI from the
 * specified PHY register. This function is called by all PHY interfaces.
 *
 * @param handle   PHY device handle.
 * @param phyReg   The PHY register.
 * @param dataPtr  The address to store the data read from the PHY register.
 * @retval kStatus_Success  PHY read success
 * @retval kStatus_PHY_SMIVisitTimeout  PHY SMI visit time out
 */
status_t PHY_Driver_Read(phy_handle_t *handle, uint32_t phyReg, uint32_t *dataPtr);

/*!
 * @brief Gets the PHY auto-negotiation status.
 *
 * @param handle   PHY device handle.
```

```
 * @param status   The auto-negotiation status of the PHY.
 *         - true the auto-negotiation is over.
 *         - false the auto-negotiation is on-going or not started.
 * @retval kStatus_Success   PHY gets status success
 * @retval kStatus_PHY_SMIVisitTimeout  PHY SMI visit time out
 */
status_t PHY_Driver_GetAutoNegotiationStatus(phy_handle_t *handle, bool *status);

/*!
 * @brief Gets the PHY link status.
 *
 * @param handle   PHY device handle.
 * @param status   The link up or down status of the PHY.
 *         - true the link is up.
 *         - false the link is down.
 * @retval kStatus_Success   PHY gets link status success
 * @retval kStatus_PHY_SMIVisitTimeout  PHY SMI visit time out
 */
status_t PHY_Driver_GetLinkStatus(phy_handle_t *handle, bool *status);

/*!
 * @brief Gets the PHY link speed and duplex.
 *
 * @brief This function gets the speed and duplex mode of PHY. User can give one of speed
 * and duplex address paramter and set the other as NULL if only wants to get one of them.
 *
 * @param handle   PHY device handle.
 * @param speed    The address of PHY link speed.
 * @param duplex   The link duplex of PHY.
 * @retval kStatus_Success   PHY gets link speed and duplex success
 * @retval kStatus_PHY_SMIVisitTimeout  PHY SMI visit time out
 */
status_t PHY_Driver_GetLinkSpeedDuplex(phy_handle_t *handle, phy_speed_t *speed, phy_duplex_t *duplex);

/*!
```

```
 * @brief Sets the PHY link speed and duplex.
 *
 * @param handle   PHY device handle.
 * @param speed    Specified PHY link speed.
 * @param duplex   Specified PHY link duplex.
 * @retval kStatus_Success   PHY gets status success
 * @retval kStatus_PHY_SMIVisitTimeout  PHY SMI visit time out
 */
status_t PHY_Driver_SetLinkSpeedDuplex(phy_handle_t *handle, phy_speed_t speed, phy_duplex_t duplex);

/*!
 * @brief Enables/disables PHY loopback.
 *
 * @param handle   PHY device handle.
 * @param mode     The loopback mode to be enabled, please see "phy_loop_t".
 * All loopback modes should not be set together, when one loopback mode is set
 * another should be disabled.
 * @param speed    PHY speed for loopback mode.
 * @param enable   True to enable, false to disable.
 * @retval kStatus_Success  PHY loopback success
 * @retval kStatus_PHY_SMIVisitTimeout  PHY SMI visit time out
 */
status_t PHY_Driver_EnableLoopback(phy_handle_t *handle, phy_loop_t mode, phy_speed_t speed, bool enable);
```

代码使用

经过改造后的代码，既可以适用于**LAN8720芯片**，也可以配合**PHY_KSZ8081芯片**工作，关键在于通过声明宏定义来选择使能何种芯片的驱动。

1. 使能LAN8720芯片:

声明宏定义 `PHY_LAN8720A`

2. 使能PHY_KSZ8081芯片:

   同时声明宏定义 `PHY_KSZ8081` 和 `FSL_FEATURE_PHYDriver_USE_RMII50M_MODE`