

Replace the LWIP library

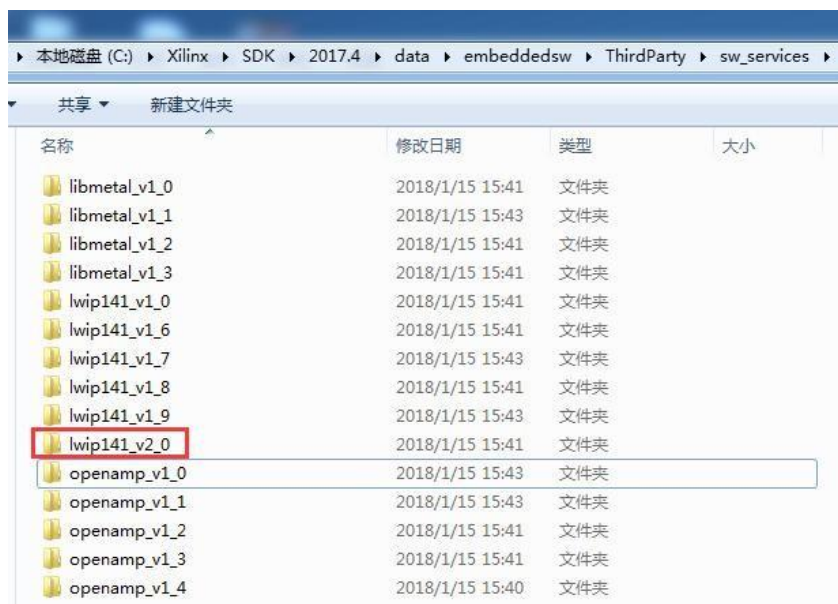
Since the built-in LWIP library can only recognize part of the PHY chip, the PHY chip used in our development board (KSZ9031 and JL2121) is not within the default support range, so the library file must be modified. You can also replace the existing library directly with the modified library. We provide modified versions of the LWIP library **2017.4** and **2020.1**, which can replace the existing library directly.

lwip例程专用库文件.zip

Locate the library file directory of software installation:

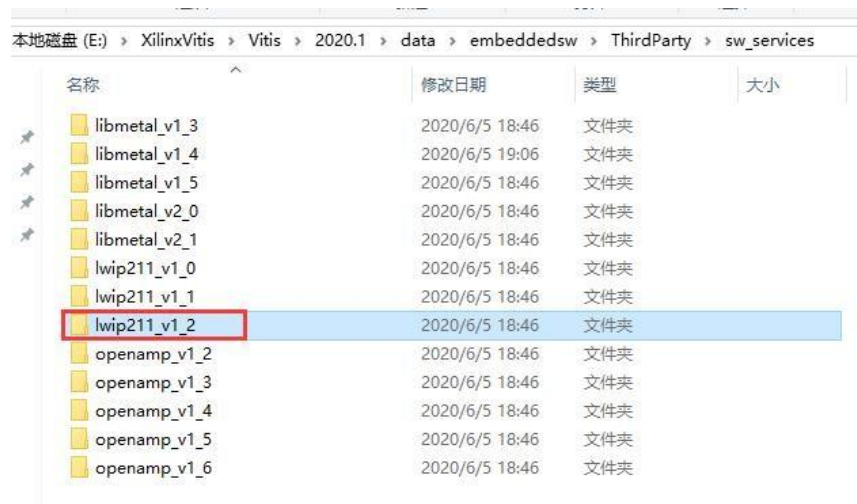
“X:\Xilinx\SDK\2017.4\data\embeddedsw\ThirdParty\sw_services”

2017.4 version is lwip141_v2_0



The path for **2020.1** version is “X:\xxx\Vitis\2020.1\data\embeddedsw\ThirdParty\sw_services”

The lwip library version is lwip211_v1_2



Extract the modified LWIP library to this folder and replace the original LWIP library.

Modify the LWIP library

If the software version is not the same as the above, you need to find the relevant C file to modify the library.

Find the files "xaxiemacif_physpeed.c" and "xemacpsif_physpeed.c" in the "lwipxxx_vx_x\src\contrib\ports\xilinx\netif", they need to be modified.

The path of 2017.4:

名称	修改日期	类型	大小
xadapter.c	2017/12/14 9:27	C Source file	7 KB
xaxiemacif.c	2017/12/14 9:27	C Source file	15 KB
xaxiemacif_dma.c	2017/12/14 9:27	C Source file	30 KB
xaxiemacif_fifo.c	2017/12/14 9:27	C Source file	12 KB
xaxiemacif_fifo.h	2017/12/14 9:27	C++ Header file	1 KB
xaxiemacif_hw.c	2017/12/14 9:27	C Source file	5 KB
xaxiemacif_hw.h	2017/12/14 9:27	C++ Header file	2 KB
xaxiemacif_physpeed.c	2017/12/14 9:27	C Source file	28 KB
xemacliteif.c	2017/12/14 9:27	C Source file	24 KB
xemacpsif.c	2017/12/14 9:27	C Source file	13 KB
xemacpsif_dma.c	2017/12/14 9:27	C Source file	27 KB
xemacpsif_hw.c	2017/12/14 9:27	C Source file	8 KB
xemacpsif_hw.h	2017/12/14 9:27	C++ Header file	2 KB
xemacpsif_physpeed.c	2017/12/14 9:27	C Source file	34 KB
xpqueue.c	2017/12/14 9:27	C Source file	3 KB

The path of 2020.1:

名称	修改日期	类型	大小
xadapter.c	2020/5/28 7:54	C 文件	12 KB
xaxiemacif.c	2020/5/28 7:54	C 文件	18 KB
xaxiemacif_dma.c	2020/5/28 7:54	C 文件	30 KB
xaxiemacif_fifo.c	2020/5/28 7:54	C 文件	12 KB
xaxiemacif_fifo.h	2020/5/28 7:54	H 文件	2 KB
xaxiemacif_hw.c	2020/5/28 7:54	C 文件	5 KB
xaxiemacif_hw.h	2020/5/28 7:54	H 文件	2 KB
xaxiemacif_mcdma.c	2020/5/28 7:54	C 文件	24 KB
xaxiemacif_physpeed.c	2020/5/28 7:54	C 文件	25 KB
xemac_jeee_reg.h	2020/5/28 7:54	H 文件	5 KB
xemacliteif.c	2020/5/28 7:54	C 文件	24 KB
xemacpsif.c	2020/5/28 7:54	C 文件	20 KB
xemacpsif_dma.c	2020/5/28 7:54	C 文件	27 KB
xemacpsif_hw.c	2020/5/28 7:54	C 文件	9 KB
xemacpsif_hw.h	2020/5/28 7:54	H 文件	2 KB
xemacpsif_physpeed.c	2020/5/28 7:54	C 文件	37 KB
xpqueue.c	2020/5/28 7:54	C 文件	3 KB

“xaxiemacif_physpeed.c” is the library file required by PL end’s axi ethernet ip, “xemacpsif_physpeed.c” is the library file required by PS end Ethernet. The two library files need to be modified.

- (1) Modify the "xemacpsif_physpeed.c" file on the PS end to add the macro definition.

```
#define MICREL_PHY_IDENTIFIER 0x22
#define MICREL_PHY_KSZ9031_MODEL 0x220

#define JLSEMI_IDENTIFIER 0x937C
#define JLSEMI_PHY_SELECT_REG_OFFSET 0x1F
#define JLSEMI_PHY_SPECIFIC_STATUS_REG_OFFSET 0x1A
#define JLSEMI_PHY_SPECIFIC_PAGE 0xA43
#define JLSEMI_PHY_LCR_PAGE 0xD04
#define JLSEMI_PHY_LED_BLINK_PAGE 0x1000
#define JLSEMI_PHY_LED_CONTROL_REG_OFFSET 0x10
#define JLSEMI_PHY_LED_BLINK_REG_OFFSET 0x14
```

Add phy speed acquisition functions for KSZ9031 and JL2121.

```
static u32_t get_phy_speed_ksz9031(XEmacPs *xemacpsp, u32_t phy_addr)
{
    u16_t temp;
    u16_t control;
    u16_t status;
    u16_t status_speed;
    u32_t timeout_counter = 0;
    u32_t temp_speed;
    u32_t phyregtemp;

    xil_printf("Start PHY autonegotiation \r\n");

    XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 2);
    XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_MAC, &control);
    control |= IEEE_RGMII_TXRX_CLOCK_DELAYED_MASK;
    XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_CONTROL_REG_MAC, control);

    XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);

    XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_AUTONEGO_ADVERTISE_REG, &control);
    control |= IEEE_ASYMMETRIC_PAUSE_MASK;
    control |= IEEE_PAUSE_MASK;
    control |= ADVERTISE_100;
    control |= ADVERTISE_10;
    XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_AUTONEGO_ADVERTISE_REG, control);

    XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_1000_ADVERTISE_REG_OFFSET,
                    &control);
    control |= ADVERTISE_1000;
    XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_1000_ADVERTISE_REG_OFFSET,
                    control);

    XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);
    XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_COPPER_SPECIFIC_CONTROL_REG,
                    &control);
```

```

control |= (7 << 12); /* max number of gigabit attempts */
control |= (1 << 11); /* enable downshift */
XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_COPPER_SPECIFIC_CONTROL_REG,
                                                         control);

XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
control |= IEEE_CTRL_AUTONEGOTIATE_ENABLE;
control |= IEEE_STAT_AUTONEGOTIATE_RESTART;
XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);

XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
control |= IEEE_CTRL_RESET_MASK;
XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);

while(1){
    XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
    if(control & IEEE_CTRL_RESET_MASK)
        continue;
    else
        break;
}

XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_STATUS_REG_OFFSET, &status);

xil_printf("Waiting for PHY to complete autonegotiation.\r\n");

while(!(status & IEEE_STAT_AUTONEGOTIATE_COMPLETE)){
    sleep(1);
    XEmacPs_PhyRead(xemacpsp, phy_addr,
                    IEEE_COPPER_SPECIFIC_STATUS_REG_2, &temp);
    timeout_counter++;

    if(timeout_counter == 30){
        xil_printf("Auto negotiation error \r\n");
        return;
    }
    XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_STATUS_REG_OFFSET, &status);
}
xil_printf("autonegotiation complete \r\n");

XEmacPs_PhyRead(xemacpsp, phy_addr, 0x1f,
                &status_speed);

if((status_speed & 0x40) == 0x40) /* 1000Mbps */
    return 1000;
elseif((status_speed & 0x20) == 0x20) /* 100Mbps */
    return 100;
elseif((status_speed & 0x10) == 0x10) /* 10Mbps */
    return 10;
else

```

```

        return 0;

    return XST_SUCCESS;
}

static u32_t get_phy_speed_JL2121(XEmacPs *xemacpsp, u32_t phy_addr)
{
    u16_t temp;
    u16_t control;
    u16_t status;
    u16_t status_speed;
    u32_t timeout_counter = 0;
    u32_t temp_speed;
    u32_t phyregtemp;

    xil_printf("phy is JL2121!\r\n");

    xil_printf("Start PHY autonegotiation \r\n");

    XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
    control |= IEEE_CTRL_RESET_MASK;
    XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);

    usleep(10000);

    XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_AUTONEGO_ADVERTISE_REG, &control);
    control |= IEEE_ASYMMETRIC_PAUSE_MASK;
    control |= IEEE_PAUSE_MASK;
    control |= ADVERTISE_100;
    control |= ADVERTISE_10;
    XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_AUTONEGO_ADVERTISE_REG, control);

    XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_1000_ADVERTISE_REG_OFFSET,
                    &control);
    control |= ADVERTISE_1000;
    XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_1000_ADVERTISE_REG_OFFSET,
                    control);

    XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
    control |= IEEE_CTRL_AUTONEGOTIATE_ENABLE;
    control |= IEEE_STAT_AUTONEGOTIATE_RESTART;
    XEmacPs_PhyWrite(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);

    while (1) {
        XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
        if (control & IEEE_CTRL_RESET_MASK)
            continue;
        else

```

```

        break;
    }

    XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_STATUS_REG_OFFSET, &status);

    xil_printf("Waiting for PHY to complete autonegotiation.\r\n");

    while ( !(status & IEEE_STAT_AUTONEGOTIATE_COMPLETE) ) {
        sleep(1);

        timeout_counter++;

        if (timeout_counter == 30) {
            xil_printf("Auto negotiation error \r\n");
            return;
        }
        XEmacPs_PhyRead(xemacpsp, phy_addr, IEEE_STATUS_REG_OFFSET, &status);
    }
    xil_printf("autonegotiation complete \r\n");

    XEmacPs_PhyWrite(xemacpsp, phy_addr, JLSEMI_PHY_SELECT_REG_OFFSET, JLSEMI_PHY_SPECIFIC_PAGE);
    XEmacPs_PhyRead(xemacpsp, phy_addr, JLSEMI_PHY_SPECIFIC_STATUS_REG_OFFSET, &status_speed);

    XEmacPs_PhyWrite(xemacpsp, phy_addr, JLSEMI_PHY_SELECT_REG_OFFSET, JLSEMI_PHY_LCR_PAGE);
    XEmacPs_PhyWrite(xemacpsp, phy_addr, JLSEMI_PHY_LED_CONTROL_REG_OFFSET, 0xAE01);

    XEmacPs_PhyWrite(xemacpsp, phy_addr, JLSEMI_PHY_SELECT_REG_OFFSET, JLSEMI_PHY_LED_BLINK_PAGE);
    XEmacPs_PhyWrite(xemacpsp, phy_addr, JLSEMI_PHY_LED_BLINK_REG_OFFSET, 0x0704);
    XEmacPs_PhyWrite(xemacpsp, phy_addr, JLSEMI_PHY_SELECT_REG_OFFSET, 0);
    if ( (status_speed & 0x20) == 0x20)/* 1000Mbps */
        return 1000;
    else if ( (status_speed & 0x10) == 0x10)/* 100Mbps */
        return 100;
    else if ( (status_speed & 0x30) == 0x0)/* 10Mbps */
        return 10;
    else
        return 0;
    return XST_SUCCESS;
}

```

Modify the function "get_IEEE_phy_speed" to add support for KSZ9031 and JL2121 (version 2017.4)

```
static u32_t get_IEEE_phy_speed(XEmacPs *xemacpsp, u32_t phy_addr)
{
    u16_t phy_identity;
    u32_t RetStatus;

    XEmacPs_PhyRead(xemacpsp, phy_addr, PHY_IDENTIFIER_1_REG,
                    &phy_identity);

    if(phy_identity == MICREL_PHY_IDENTIFIER)
    {
        RetStatus = get_phy_speed_ksz9031(xemacpsp, phy_addr);
    }else if (phy_identity == JLSEMI_IDENTIFIER) {
        RetStatus = get_phy_speed_JL2121(xemacpsp, phy_addr);
    }
    else if (phy_identity == PHY_TI_IDENTIFIER) {
        RetStatus = get_TI_phy_speed(xemacpsp, phy_addr);
    } else {
        RetStatus = get_Marvell_phy_speed(xemacpsp, phy_addr);
    }

    return RetStatus;
}
```


Version 2020.1 are as follows:

```
static u32_t get_IEEE_phy_speed(XEmacPs *xemacpsp, u32_t phy_addr)
{
    u16_t phy_identity;
    u32_t RetStatus;

    XEmacPs_PhyRead(xemacpsp, phy_addr, PHY_IDENTIFIER_1_REG,
                    &phy_identity);
    if(phy_identity == MICREL_PHY_IDENTIFIER) {
        RetStatus = get_phy_speed_ksz9031(xemacpsp, phy_addr);

    }else if (phy_identity == JLSEMI_IDENTIFIER) {
        RetStatus = get_phy_speed_JL2121(xemacpsp, phy_addr);
    } else if (phy_identity == PHY_TI_IDENTIFIER) {
        RetStatus = get_TI_phy_speed(xemacpsp, phy_addr);
    } else if (phy_identity == PHY_REALTEK_IDENTIFIER) {
        RetStatus = get_Realtek_phy_speed(xemacpsp, phy_addr);
    } else {
        RetStatus = get_Marvell_phy_speed(xemacpsp, phy_addr);
    }

    return RetStatus;
}
```


(2) Modify the "xaxiemacif_physpeed.c" file on the PL end and add related macro definition.

```
#define MICREL_PHY_IDENTIFIER 0x22
#define MICREL_PHY_KSZ9031_MODEL 0x220

#define JLSEMI_IDENTIFIER 0x937C
#define JLSEMI_PHY_SELECT_REG_OFFSET 0x1F
#define JLSEMI_PHY_SPECIFIC_STATUS_REG_OFFSET 0x1A
#define JLSEMI_PHY_SPECIFIC_PAGE 0xA43
#define JLSEMI_PHY_LCR_PAGE 0xD04
#define JLSEMI_PHY_LED_BLINK_PAGE 0x1000
#define JLSEMI_PHY_LED_CONTROL_REG_OFFSET 0x10
#define JLSEMI_PHY_LED_BLINK_REG_OFFSET 0x14
```

Add phy speed acquisition functions for KSZ9031 and JL2121.

```
Unsigned int get_phy_speed_ksz9031(XAxiEthernet *xaxiemacp, u32 phy_addr)
{
    u16 control;
    u16 status;
    u16 partner_capabilities;
    xil_printf("Start PHY autonegotiation \r\n");

    XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 2);
    XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_CONTROL_REG_MAC, &control);
    //control |= IEEE_RGMII_TXRX_CLOCK_DELAYED_MASK;
    control &=~(0x10);
    XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_CONTROL_REG_MAC, control);

    XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);

    XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_AUTONEGO_ADVERTISE_REG, &control);
    control |= IEEE_ASYMMETRIC_PAUSE_MASK;
    control |= IEEE_PAUSE_MASK;
    control |= ADVERTISE_100;
    control |= ADVERTISE_10;
    XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_AUTONEGO_ADVERTISE_REG, control);

    XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_1000_ADVERTISE_REG_OFFSET,
                                                                    &control);

    control |= ADVERTISE_1000;
    XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_1000_ADVERTISE_REG_OFFSET,
                                                                    control);

    XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);
    XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_COPPER_SPECIFIC_CONTROL_REG,
                                                                    &control);

    control |= (7<<12); /* max number of gigabit attempts */
    control |= (1<<11); /* enable downshift */
    XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_COPPER_SPECIFIC_CONTROL_REG,
                                                                    control);

    XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
    control |= IEEE_CTRL_AUTONEGOTIATE_ENABLE;
```

```

control |= IEEE_STAT_AUTONEGOTIATE_RESTART;

XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);

XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET,&control);
control |= IEEE_CTRL_RESET_MASK;
XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);

while(1){
    XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET,&control);
    if(control & IEEE_CTRL_RESET_MASK)
        continue;
    else
        break;
}
xil_printf("Waiting for PHY to complete autonegotiation.\r\n");

XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_STATUS_REG_OFFSET,&status);
while(!(status & IEEE_STAT_AUTONEGOTIATE_COMPLETE)){
    sleep(1);
    XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_STATUS_REG_OFFSET,
                                                                    &status);
}

xil_printf("autonegotiation complete \r\n");

XAxiEthernet_PhyRead(xaxiemacp, phy_addr,0x1f,&partner_capabilities);

if((partner_capabilities &0x40)==0x40)/* 1000Mbps */
    return1000;
elseif((partner_capabilities &0x20)==0x20)/* 100Mbps */
    return100;
elseif((partner_capabilities &0x10)==0x10)/* 10Mbps */
    return10;
else
    return0;
}

static u32_t get_phy_speed_JL2121(XAxiEthernet *xaxiemacp, u32_t phy_addr)
{
    u16_t temp;
    u16_t control;
    u16_t status;
    u16_t status_speed;
    u32_t timeout_counter = 0;
    u32_t temp_speed;
    u32_t phyregtemp;

    xil_printf("phy is JL2121!\r\n");

```

```

xil_printf("Start PHY autonegotiation \r\n");

XAXiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
control |= IEEE_CTRL_RESET_MASK;
XAXiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);

usleep(10000);

XAXiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_AUTONEGO_ADVERTISE_REG, &control);
control |= IEEE_ASYMMETRIC_PAUSE_MASK;
control |= IEEE_PAUSE_MASK;
control |= ADVERTISE_100;
control |= ADVERTISE_10;
XAXiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_AUTONEGO_ADVERTISE_REG, control);

XAXiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_1000_ADVERTISE_REG_OFFSET,
                    &control);
control |= ADVERTISE_1000;
XAXiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_1000_ADVERTISE_REG_OFFSET,
                    control);

XAXiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
control |= IEEE_CTRL_AUTONEGOTIATE_ENABLE;
control |= IEEE_STAT_AUTONEGOTIATE_RESTART;
XAXiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);

while (1) {
    XAXiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
    if (control & IEEE_CTRL_RESET_MASK)
        continue;
    else
        break;
}

XAXiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_STATUS_REG_OFFSET, &status);

xil_printf("Waiting for PHY to complete autonegotiation.\r\n");

while ( !(status & IEEE_STAT_AUTONEGOTIATE_COMPLETE) ) {
    sleep(1);

    timeout_counter++;

    if (timeout_counter == 30) {
        xil_printf("Auto negotiation error \r\n");
        return;
    }
}

```

```

        XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_STATUS_REG_OFFSET, &status);
    }
    xil_printf("autonegotiation complete \r\n");

    XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, JLSEMI_PHY_SELECT_REG_OFFSET, JLSEMI_PHY_SPECIFIC_PAGE);
    XAxiEthernet_PhyRead(xaxiemacp, phy_addr, JLSEMI_PHY_SPECIFIC_STATUS_REG_OFFSET, &status_speed);

    XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, JLSEMI_PHY_SELECT_REG_OFFSET, JLSEMI_PHY_LCR_PAGE);
    XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, JLSEMI_PHY_LED_CONTROL_REG_OFFSET, 0xAE01);

    XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, JLSEMI_PHY_SELECT_REG_OFFSET, JLSEMI_PHY_LED_BLINK_PAGE);
    XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, JLSEMI_PHY_LED_BLINK_REG_OFFSET, 0x0704);
    XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, JLSEMI_PHY_SELECT_REG_OFFSET, 0);

    u16_t txc_dly_sel = 0x0 ;
    u16_t txc_in_inverse = 0x0 ;
    u16_t rxc_dly_sel = 0x3 ;
    u16_t rgmii_rxc_dly_sel = 0x0 ;
    u16_t rxc_out_inverse = 0x0 ;

    XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, JLSEMI_PHY_SELECT_REG_OFFSET, 171);
    XAxiEthernet_PhyRead(xaxiemacp, phy_addr, 16, &control);
    control &= 0x81FF;
    control |= (txc_in_inverse<<14 & 0x4000) | (txc_dly_sel<<12 & 0x3000) | (rxc_out_inverse<<11 & 0x0800) |
(rxc_dly_sel<<9 & 0x0600) ;

    XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, 16, control);
    XAxiEthernet_PhyRead(xaxiemacp, phy_addr, 17, &control);
    control &= 0xFFFE;
    control |= rgmii_rxc_dly_sel & 0x1 ;
    XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, 17, control);

    XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, JLSEMI_PHY_SELECT_REG_OFFSET, 0);
    if ( (status_speed & 0x20) == 0x20)/* 1000Mbps */
        return 1000;
    else if ( (status_speed & 0x10) == 0x10)/* 100Mbps */
        return 100;
    else if ( (status_speed & 0x30) == 0x0)/* 10Mbps */
        return 10;
    else
        return 0;
    return XST_SUCCESS;
}

```

Modify the function " get_IEEE_phy_speed " to add support for KSZ9031 and JL2121 (version 2017.4)

```
unsigned get_IEEE_phy_speed(XAxiEthernet *xaxiemacp)
{
    u16 phy_identifier;
    u16 phy_model;
    u8 phytype;

#ifdef XPAR_AXIETHERNET_0_BASEADDR
    u32 phy_addr = detect_phy(xaxiemacp);

    /* Get the PHY Identifier and Model number */
    XAxiEthernet_PhyRead(xaxiemacp, phy_addr, PHY_IDENTIFIER_1_REG, &phy_identifier);
    XAxiEthernet_PhyRead(xaxiemacp, phy_addr, PHY_IDENTIFIER_2_REG, &phy_model);

    /* Depending upon what manufacturer PHY is connected, a different mask is
    * needed to determine the specific model number of the PHY. */
    if (phy_identifier == MARVEL_PHY_IDENTIFIER) {
        phy_model = phy_model & MARVEL_PHY_MODEL_NUM_MASK;

        if (phy_model == MARVEL_PHY_88E1116R_MODEL) {
            return get_phy_speed_88E1116R(xaxiemacp, phy_addr);
        } else if (phy_model == MARVEL_PHY_88E1111_MODEL) {
            return get_phy_speed_88E1111(xaxiemacp, phy_addr);
        }
    } else if (phy_identifier == TI_PHY_IDENTIFIER) {
        phy_model = phy_model & TI_PHY_DP83867_MODEL;
        phytype = XAxiEthernet_GetPhysicalInterface(xaxiemacp);

        if (phy_model == TI_PHY_DP83867_MODEL && phytype == XAE_PHY_TYPE_SGMII) {
            return get_phy_speed_TI_DP83867_SGMII(xaxiemacp, phy_addr);
        }

        if (phy_model == TI_PHY_DP83867_MODEL) {
            return get_phy_speed_TI_DP83867(xaxiemacp, phy_addr);
        }
    }
    else if(phy_identifier == MICREL_PHY_IDENTIFIER)
    {
        xil_printf("Phy %d is KSZ9031\n\r", phy_addr);
        return get_phy_speed_ksz9031(xaxiemacp, phy_addr);
    }
    else if(phy_identifier == JLSEMI_IDENTIFIER)
    {
        return get_phy_speed_JL2121(xaxiemacp, phy_addr);
    }
    else {
        LWIP_DEBUGF(NETIF_DEBUG, ("XAxiEthernet get_IEEE_phy_speed: Detected PHY with unknown
identifier/model.\r\n"));
    }
#endif
#ifdef PCM_PMA_CORE_PRESENT
    return get_phy_negotiated_speed(xaxiemacp, phy_addr);
#endif
}
```

```
#endif  
  
}
```

Versions 2020.1 are as follows:

```
unsigned get_IEEE_phy_speed(XAxiEthernet *xaxiemaclp)  
{  
    ul6 phy_identifier;  
    ul6 phy_model;  
    u8 phytype;  
  
#ifdef XPAR_AXIETHERNET_O_BASEADDR  
    u32 phy_addr = detect_phy(xaxiemaclp);  
  
    /* Get the PHY Identifier and Model number */  
    XAxiEthernet_PhyRead(xaxiemaclp, phy_addr, PHY_IDENTIFIER_1_REG, &phy_identifier);  
    XAxiEthernet_PhyRead(xaxiemaclp, phy_addr, PHY_IDENTIFIER_2_REG, &phy_model);  
  
    /* Depending upon what manufacturer PHY is connected, a different mask is  
    * needed to determine the specific model number of the PHY. */  
    if (phy_identifier == MARVEL_PHY_IDENTIFIER) {  
        phy_model = phy_model & MARVEL_PHY_MODEL_NUM_MASK;  
  
        if (phy_model == MARVEL_PHY_88E1116R_MODEL) {  
            return get_phy_speed_88E1116R(xaxiemaclp, phy_addr);  
        } else if (phy_model == MARVEL_PHY_88E1111_MODEL) {  
            return get_phy_speed_88E1111(xaxiemaclp, phy_addr);  
        }  
    } else if (phy_identifier == TI_PHY_IDENTIFIER) {  
        phy_model = phy_model & TI_PHY_DP83867_MODEL;  
        phytype = XAxiEthernet_GetPhysicalInterface(xaxiemaclp);  
  
        if (phy_model == TI_PHY_DP83867_MODEL && phytype == XAE_PHY_TYPE_SGMII) {  
            return get_phy_speed_TI_DP83867_SGMII(xaxiemaclp, phy_addr);  
        }  
  
        if (phy_model == TI_PHY_DP83867_MODEL) {  
            return get_phy_speed_TI_DP83867(xaxiemaclp, phy_addr);  
        }  
    }  
    else if (phy_identifier == MICREL_PHY_IDENTIFIER)  
    {  
        xil_printf("Phy %d is KSZ9031\n\r", phy_addr);  
        get_phy_speed_ksz9031(xaxiemaclp, phy_addr);  
    }  
    else if (phy_identifier == JLSEMI_IDENTIFIER)  
    {  
        return get_phy_speed_JL2121(xaxiemaclp, phy_addr);  
    }  
}
```

```

else {
    LWIP_DEBUGF(NETIF_DEBUG, ("XAxiEthernet get_IEEE_phy_speed: Detected PHY with unknown
identifier/model.\r\n"));
}
#endif
#ifdef PCM_PMA_CORE_PRESENT
    return get_phy_negotiated_speed(xaxiemaclp, phy_addr);
#endif
}

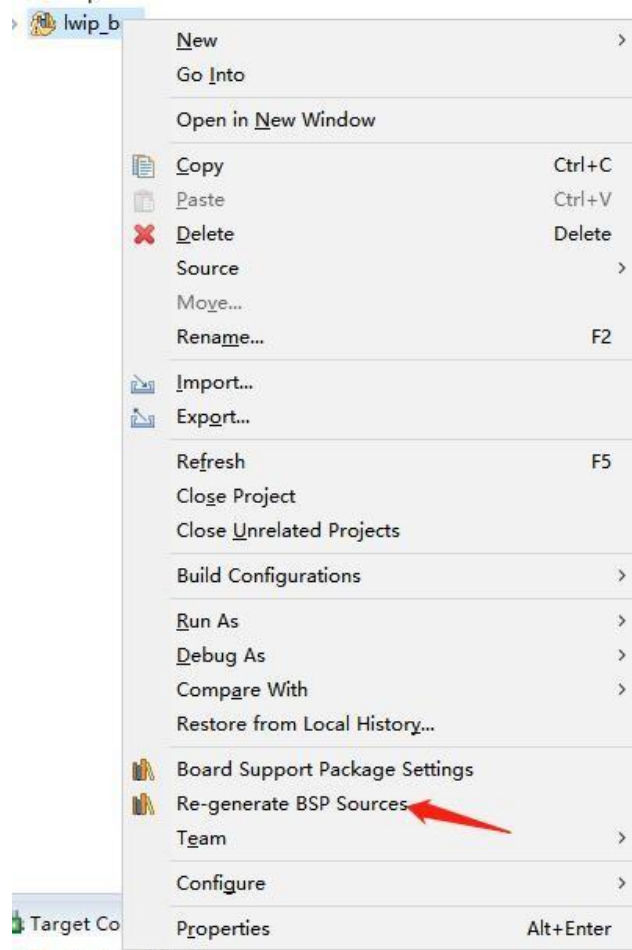
```

Update the SDK or Vitis project

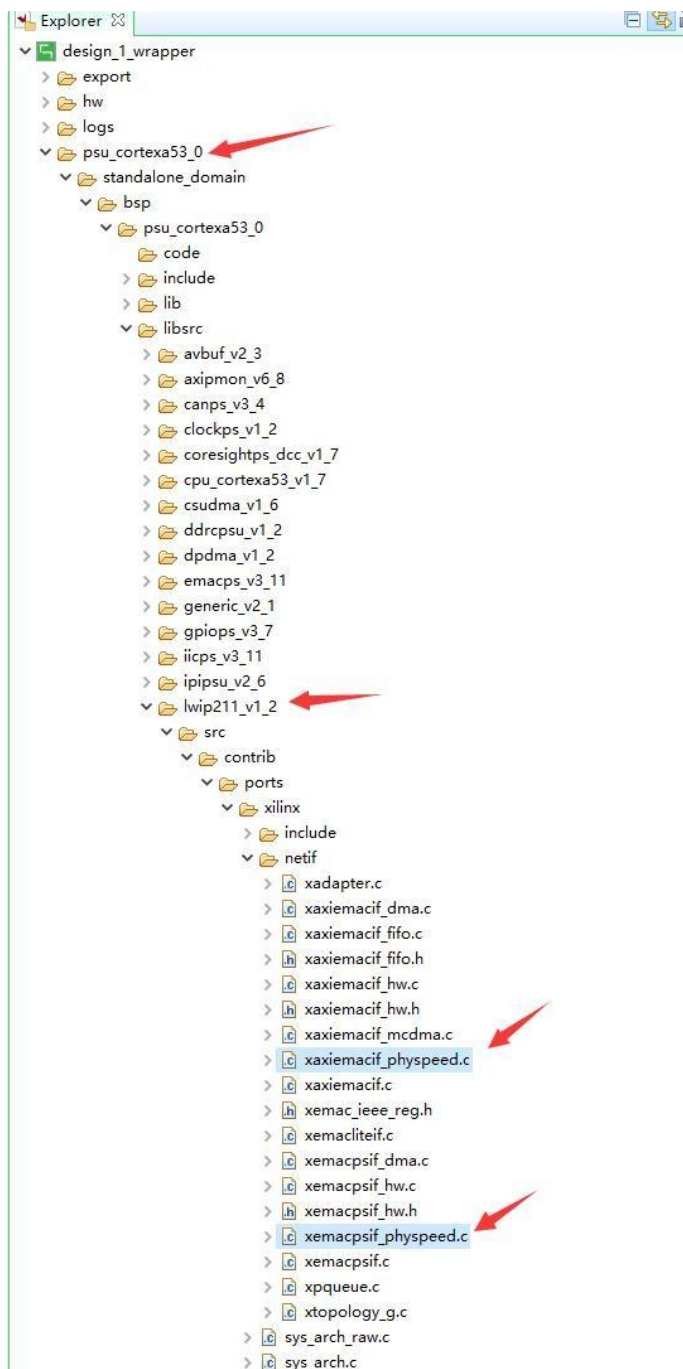
After replacing or modifying LWIP, you will need to update the SDK or Vitis project, two methods are provided below:

(1) Method of updating BSP

In the SDK of version 2017.4, select the bsp project, lwip_bsp, right-click Re-generate BSP Sources, update BSP, and BSP will re-import LWIP library from the software installation directory.

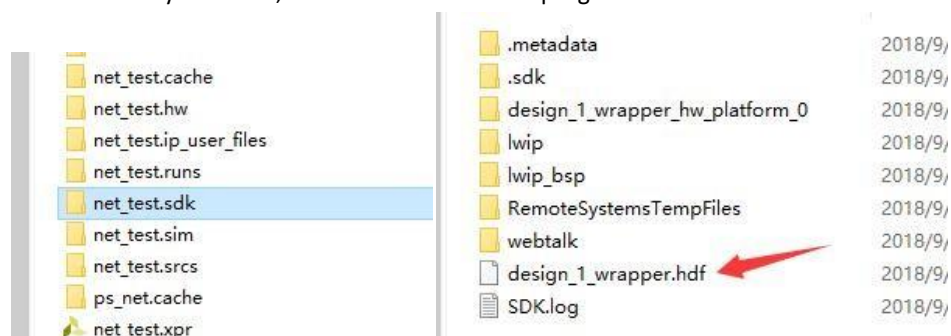


After the update, find the modified LWIP library file in BSP and confirm whether it has been modified. If you confirm that the modification is correct, you can download the program.


















(2) Re-build SDK or Vitis project

Take version 2017.4 as an example, exit the SDK software, find the xx.sdk folder under the project, keep the xx.hdf file, and delete other files. Then Launch SDK again, and create a new lwip echo server template according to the tutorial. You can also confirm whether the library file is successfully modified, and then download the program.



Take version 2020.1 as an example, exit the Vitis software, find the vitis folder, keep the auto_create_vitis and xx.xsa, delete the others, and re-create the vitis project. You can also confirm whether the library file is modified successfully, and then download the

program.

 .metadata	21
 .sdk	21
 .Xil	21
 auto_create_vitis 	21
 design_1_wrapper	21
 net_test	21
 net_test_system	21
 RemoteSystemsTempFiles	21
 tcp	21
 tcp_system	21
 .analytics	21
 design_1_wrapper.xsa 	21
 IDE.log	21