Single-Chip Solution for Hi-Speed USB2.0 (480Mbps) JTAG Debugger

1. Preface

The USB2.0 (480Mbps) to JTAG interface solution based on CH32V305/CH32V307 MCU can be used to debug or download devices such as CPU, DSP, FPGA and CPLD. Only one CH32V305/307 MCU is needed, and no CPLD and USB PHY auxiliary devices are required.

The JTAG debug programmer built in the current solution is named USB2.0_JTAG, using the custom BitBang and ByteShift protocol transmission mode. The JTAG Write speed can reach 36Mbit/s. The upper computer software used is OpenOCD. After the USB2.0_JTAG interface is added in OpenOCD, OpenOCD then supports the operation on USB2.0_JTAG, so as to realize the JTAG download and debug functions. The target FPGA used for downloading in this solution is XILINX SPARTAN-6 XC6SLX9, and the download instructions are as follows:

```
openocd.exe -f ./usb20jtag.cfg -f ../scripts/cpld/xilinx-xc6s.cfg -c
"init;xc6s_program xc6s.tap; pld load 0 ./led.bit" -c exit
```

The download file and the download completion prompt are as follow (the size of the *led.bit* download file is 333KB):

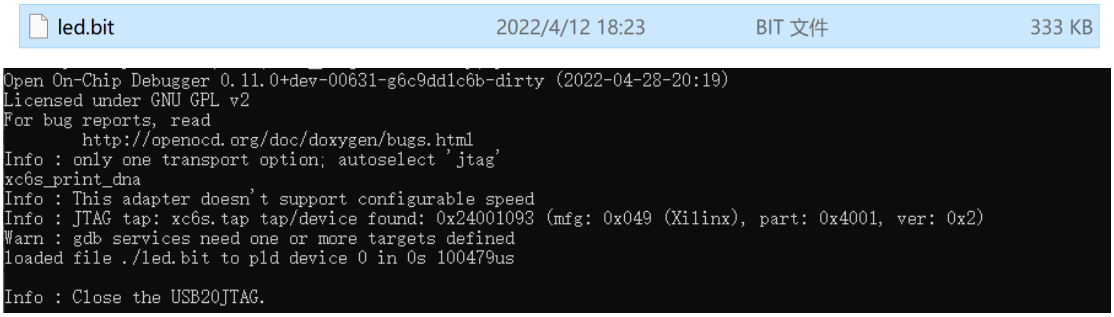

Figure 1 USB2.0_Jtag download

Jtag Debugger	Appearance	Size of file	Time to download	Speed
USB2.0_Jtag		333KB	0.100s	3.33MB/s

The block diagram of the solution structure is as follows:

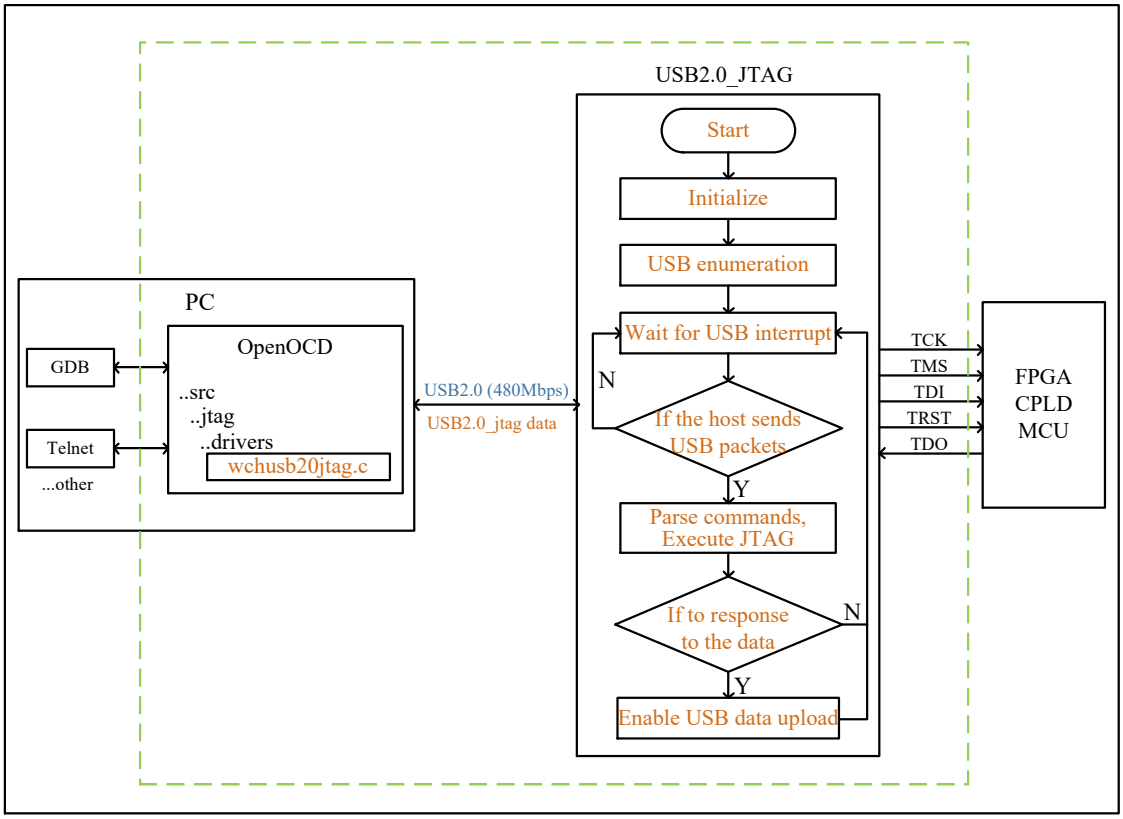

Figure 2 JTAG debugger solution based on CH32V305/CH32V307

The complete development resources are available, including reference schematic diagram, MCU program source codes, USB2.0 (480Mbps) high-speed device general driver, USB to JTAG/SPI function library source codes, reference program source codes, and communication protocol.

The hardware and software environments required to implement this solution are:

Hardware: CH32V305/307EVT evaluation board
Software: OpenOCD source codes
Compiler: Cygwin (Refer to Section 3.3 for the installation of the compiler)
FPGA target board: XILINX SPARTAN-6 XC6SLX9

Links to download MCU-related materials:

MCU IDE (MounRiver): http://www.mounriver.com/download
MCU ISP Tool: http://www.wch-ic.com/downloads/WCHISPTool_Setup_exe.html
USB driver: http://www.wch-ic.com/downloads/CH372DRV_EXE.html

2. Hardware

This solution is tested using the CH32V305/307EVT evaluation board. The CH32V305/307 is an industrial-grade general-purpose MCU based on Qingke 32-bit RISC-V. Equipped with a hardware stack and fast interrupt entry, the interrupt response speed is greatly improved on the basis of standard RISC-V. Equipped with V4F core, the system clock frequency can be up to 144MHz. Independent GPIO power supply. Built-in two 12-bit ADC modules, two 12-bit DAC modules, several timers, multi-channel capacitive touch-key detection (TKEY) and other functions. It also provides standard and dedicated communication interfaces: I2C, I2S, SPI, USART, SDIO, CAN

controller, USB2.0 full-speed host/device controller, USB2.0 (480Mbps) high-speed host/device controller (built-in self-developed PHY transceiver, no three-party IP cost, so the cost is similar to that of full-speed USB), digital video port, Gigabit Ethernet controller, etc. The system clock frequency can be 144MHz. The computing power is higher and the processing speed is faster. The devices with more than 64 pins support GPIO independent power supply (support 1.8, 2.5 and 3.3V). In addition, both the USB interface and the SPI interface have hardware DMA, which is also the key to the faster speed of this solution.

The related resources of CH32V305/307 can be obtained at: http://www.wch-ic.com/.

Definitions of used pins:

CH32V305/CH32V307 pin	USB2.0_JTAG pin
PB6	TRST (optional)
PB12	TMS
PB13	TCK
PB14	TDO
PB15	TDI

2.1. Firmware structure

2.1.1. Download USB data

When the CH32V305/307 is initialized, enable the high-speed USB2.0 (480Mbps) interface, and set the endpoint size to 512 bytes. In order to improve the download and processing efficiency, a 4096-byte ring buffer is allocated for USB download, making sure that you can continue to receive subsequent USB data while processing the previously downloaded data.

Download procedure:
Step 1: The computer downloads a USB packet, and the MCU generates an endpoint OUT successful interrupt.
Step 2: The MCU switches the next received DMA address of the downlink endpoint (only switches the DMA address, no data copying is required), and calculates the relevant variables.
Step 3: The MCU judges whether there is enough buffer to receive the next packet of data. If the buffer is enough, the endpoint is set to allow continuing to receive data. If the buffer is insufficient, the endpoint is set to suspend receiving data. Only after the previous data is processed, there is enough buffer to set the endpoint again to allow to continue to receive data.
Step 4: Continue to repeat steps 1-3.

2.1.2. Parse commands

The JTAG interface has a total of 5 commands, as shown in the table below:

0xD0	JTAG interface initialization command (DEF_CMD_JTAG_INIT)
0xD1	JTAG interface pin bit control command (DEF_CMD_JTAG_BIT_OP)
0xD2	JTAG interface pin bit control and read command (DEF_CMD_JTAG_BIT_OP_RD)

0xD3	JTAG interface data shift command (DEF_CMD_JTAG_DATA_SHIFT）
0xD4	JTAG interface data shift and read command (DEF_CMD_JTAG_DATA_SHIFT_RD)

Format of commands:

CMD	LEN	DATA
Command code	Length of subsequent data	Subsequent data
1-byte	2-byte	N-byte (0<=N<=507)

General description of commands:

1) D0 command is used to initialize the mode and speed of JTAG.
2) D1 command is used to control the TCK, TMS, TDI and TRST pins. 1-byte data can control the four pins to output different levels, mainly used for the switching of the JTAG state machine.
3) D2 command is also used to control the TCK, TMS, TDI and TRST pins, as well as to read the data on the TDO pin while controlling the output levels of the 4 pins. The D2 command is mainly used to read the data returned by the target device from TDO when the state machine switches to SHIFT-I(D)R to perform IR or DR write operations.
4) D3 command is used to transmit the data to be sent on TDI. When using the D3 command, the TMS and TRST pins cannot be controlled. The CH32V305 controls and generates 8 cycles of TCK. TDI comes from the byte data transmitted by the D3 command. The D3 command is mainly used to write IR or DR in batches through TDI in the SHIFT-I(D)R state.
5) The functions of the D4 command and the D3 command are basically the same, the difference is that the data returned by the target device from the TDO pin is read while transferring data in batches.

Since the last bit of data is captured on the rising edge that enters the Exit1-I(D)R state when writing to IR or DR, if you want to write N-bit data to I(D)R, in SHIFT-I(D)R state, (N/8) bytes of data can be written through the D3(D4) command, the remaining (N%8 - 1) bits of data can be written through the D1(D2) command, and the last bit of data switches to the Exit1-I(D)R state through the D1(D2) command while writing to I(D)R.

In order to improve the command download and processing efficiency, the computer can combine several commands into one USB packet to download, or download several consecutive USB packets, and the MCU can automatically divide the packets.

Procedure:

Step 1: The MCU judges whether there is any remaining USB download data that has not been processed. If not, it continues to wait. If so, it takes out a packet of data (a single packet of up to 512 bytes) for processing.

Step 2: The MCU analyzes and processes the extracted data packets in turn, and performs relative operations. If there is readback data, it is stored in the corresponding JTAG receive buffer. One USB packet can contain several commands, and one command can span 2 USB packets.

Step 3: The MCU switches the buffer processing pointers and calculates the relevant variables.
Step 4: Continue to repeat steps 1-3.

2.1.3. Upload USB data

When a D2 or D4 command is executed, the returned data is read from the target device and stored in the receive buffer. The receive buffer is a 4096-byte ring buffer, so that the data upload is not affected while storing the data.

Upload procedure:
Step 1: When the MCU executes the D2 or D4 command, it reads back the data returned by the target device and stores it in the receive buffer.
Step 2: After the MCU executes this command, it determines whether there is readback data to be uploaded. If there is no data to be uploaded, it continues to execute the next command. If there is readback data to be uploaded, set the endpoint upload DMA address and enable USB data upload (Only need to set DMA address, no need to copy data).
Step 3: The MCU waits for the data to be uploaded successfully. If the upload is successful, it calculates the relevant variables. Otherwise, it cancels the data upload after waiting for a timeout, so that the remaining data can be uploaded next time.
Step 4: Continue to repeat steps 1-3.

2.2. Parse part of the codes

This section mainly introduces the code implementation of D0-D4 commands.
(1). D1 (JTAG interface pin bit control command):

```
void JTAG_Port_BitShift( UINT8 dat )
{
    PIN_TDI_OUT( 0 != ( dat & DEF_TDI_BIT_OUT ) );
    PIN_TMS_OUT( 0 != ( dat & DEF_TMS_BIT_OUT ) );
    PIN_TCK_OUT( 0 != ( dat & DEF_TCK_BIT_OUT ) );
    PIN_TRST_OUT( 0 != ( dat & DEF_TRST_BIT_OUT ) );
}
```

```
#define DEF_TRST_BIT_OUT          ( 0x20 )
#define DEF_TDI_BIT_OUT           ( 0x10 )
#define DEF_NCS_BIT_OUT           ( 0x08 )
#define DEF_NCE_BIT_OUT           ( 0x04 )
#define DEF_TMS_BIT_OUT           ( 0x02 )
#define DEF_TCK_BIT_OUT           ( 0x01 )
#define DEF_TDO_BIT_IN            ( 0 )
```

1-byte data can control the level of the TDI, TMS, TCK, TRST pins by setting the corresponding

bits of the pins. 2 bytes can complete one clock cycle. For example, 0x13 means that the TDI, TMS, TCK pins are all set to high, and the TRST pin is set to low.

(2). D2 (JTAG interface pin bit control and read command):

The D2 command also uses the JTAG_Port_BitShift function. The difference is that when it is judged that the TCK pin corresponding to the downloaded data is high, the TDO pin level will be read and put into the upload buffer.

(3). D3 (JTAG interface data shift command):

```
JTAG_Port_SwTo_SPIMode( );

if( count == DEF_HS_PACK_MAX_LEN )
{
    for( i = 0; i < 10; i++ )
    {
        SPI2->DATAR = *pTxbuf++;
        while( ( SPI2->STATR & SPI_I2S_FLAG_TXE ) == RESET );
        SPI2->DATAR = *pTxbuf++;
        while( ( SPI2->STATR & SPI_I2S_FLAG_TXE ) == RESET );
        SPI2->DATAR = *pTxbuf++;
        while( ( SPI2->STATR & SPI_I2S_FLAG_TXE ) == RESET );
```

This command is used to transmit TDI data in batches. To improve data transmission efficiency, SPI mode is selected for data transmission, and TCK is generated by SPI. When executing the D3 command, you first need to configure the JTAG pin to SPI mode, then enable the SPI, and finally start sending data. If the transmission length (within 1 packet of USB data) is the maximum length of 507, the fast processing mode is selected to send data (exchange code overhead for execution speed, execute 10 loops, send 50 data per loop, and finally send the remaining 7 data). If the D3 command transmits less than 507 data in a USB packet, it is sent in the normal processing mode (execute the while loop to send data).

```
while( SPI2->STATR & SPI_I2S_FLAG_BSY );

JTAG_Port_SwTo_GPIOMode( );
```

After all data is transmitted, the SPI pin is reset to JTAG mode after judging the data transmission end flag, and the SPI is disabled.

(4). D4 (JTAG interface data shift and read command):

```
UINT8 JTAG_Port_DataShift_RD( UINT8 dat )
{
    UINT32 dout = dat;
    UINT32 din;

#define JTGA_SHIFT_BIT( )    PIN_TDI_OUT( dout & 0x01 ); din = PIN_TDO_IN( ); PIN_TCK_1( );\
                             dout = ( dout >> 1 ) | ( din << 7 ); PIN_TCK_0( )

    JTGA_SHIFT_BIT( );
    JTGA_SHIFT_BIT( );
    JTGA_SHIFT_BIT( );
    JTGA_SHIFT_BIT( );
    JTGA_SHIFT_BIT( );
    JTGA_SHIFT_BIT( );
    JTGA_SHIFT_BIT( );
    JTGA_SHIFT_BIT( );

    return( dout & 0xff );
}
```

The D4 command uses the method of manually setting TCK. Each byte of data is split into 8 bits, and then 8 clock cycles is completed by pulling the TCK pin high and low. Send TDI and read the TDO pin level.

3. Software

3.1. OpenOCD introduction

OpenOCD (Open On-Chip Debugger) is a free and open-source project designed to provide debugging, in-system programming, and boundary scan testing for embedded target devices. Combining with a debug adapter, it can implement the above functions. The debug adapter is a small hardware module that provides the required electrical signals for the target hardware to be debugged. This solution is to implement a debugger based on CH32V305/CH32V307. Open the *openocd* source code directory. The main code is concentrated in the *src* directory. The code to implement read/write/erase operation on various chip with built-in FLASH is in the *flash* directory. The code for command line and command analysis is in the *helper* directory. The code for the operating systems supported by OpenOCD is in the *rtos* directory. The code for remote connection services such as GDB TELNET is in the *server* directory. The code for various target architectures supported by OpenOCD is in the *target* directory. The code to implement transport protocol layer is in the *jtag* directory. The solution below is to implement a new debugger interface in the *jtag/drivers* directory.

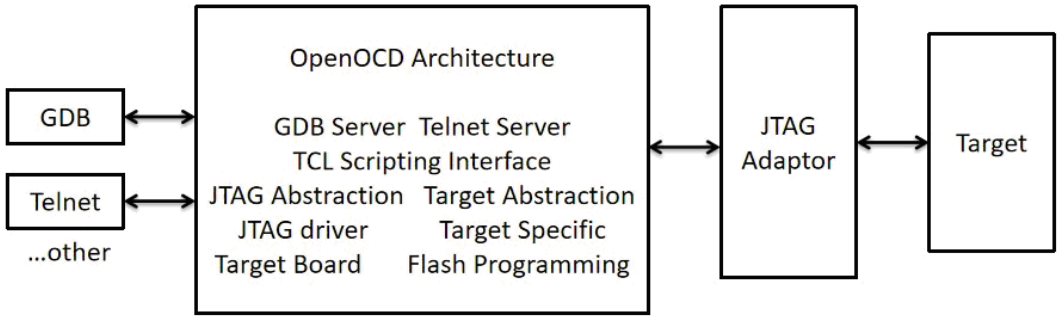

Figure 3 OpenOCD structure diagram

3.2. Implement support for USB2.0_JTAG interface in OpenOCD

3.2.1. Create a USB2.0_JTAG interface in OpenOCD

1. First, get the OpenOCD source code from: https://sourceforge.net/p/openocd/code/ci/master/tree/. Add interface support to the project in the directory.

2. Modify the *configure.ac* file in the project root directory to add driver support for USB2.0_JTAG as follows:

```
+112  [[usb20jtag], [USB20JTAG Programmer], [USB20JTAG]],
+278  AC_ARG_ENABLE([usb20jtag],
+     AS_HELP_STRING([--enable-usb20jtag], [Enable building support for
USB20JTAG]),
+     [build_usb20jtag=$enableval], [build_usb20jtag=no])
+529  AS_IF([test "x$build_usb20jtag" = "xyes"], [
+AC_DEFINE([BUILD_USB20JTAG], [1], [1 if you want USB20JTAG.])
+     ], [
+AC_DEFINE([BUILD_USB20JTAG], [0], [0 if you don't want USB20JTAG.])
+])
+722  AM_CONDITIONAL([USB20JTAG], [test "x$build_usb20jtag" = "xyes"])
```

3. Modify the *interface.c* file in the *src/jtag* project directory to add the usb20jtag_adapter_driver debugger, as follows:

```
+153    #if BUILD_USB20JTAG ==1
+       externstruct adapter_driver usb20jtag_adapter_driver;
+       #endif
+269    #if BUILD_USB20JTAG ==1
+     &usb20jtag_adapter_driver,
+       #endif
```

4. Modify the *Makefile.am* file in the project root directory to add compilation support, as follows:

```
+188    if USB20JTAG
+       DRIVERFILES += %D%/usb20jtag.c
```

3.2.2. Build USB2.0_JTAG interface code

In the *jtag/drivers* directory, add the *usb20jtag.c* file. To build the code of USB2.0_JTAG interface, you can refer to the following operations.

3.2.3. USB2.0_JTAG protocol mechanism

The packet header command code and packet length setting code for the USB2.0_JTAG built with CH32V305 are added based on BitBang and Byteshift modes. Refer to Section 2.1.2 for the command code format and protocol transmission format.

3.2.4. Interface entry API

The registration of the usb20jtag_adapter_driver debugger structure is completed in Section 3.2.1. The internal structure is as follows:

```
struct adapter_driver usb20jtag_adapter_driver = {
  .name = "usb20jtag",                          // 接口驱动名称
  .transports = jtag_only,                      // 仅支持 JTAG 调试
  .commands = usb20jtag_command_handlers,       // 命令处理函数

  .init = usb20jtag_init,                       // USB2.0_JTAG 初始化函数
  .quit = usb20jtag_quit,                       // USB2.0_JTAG 退出函数
  .jtag_ops = &usb20jtag_interface,             // JTAG 接口的调试输入 API
};
```

The corresponding interface function (usb20jtag_interface) is as follows:

```
static struct jtag_interface usb20jtag_interface = {
  .supported = DEBUG_CAP_TMS_SEQ,
  .execute_queue = usb20jtag_execute_queue,     // 由 JTAG 驱动层调用
};
```

The driver layer in OpenOCD can initialize and exit the device by calling usb20jtag_adapter_driver. In the usb20jtag_interface function, the usb20jtag_execute_queue execution command queue called by the JTAG protocol layer is defined, and the corresponding operation is completed by the CMD data sent by the JTAG protocol layer. So we only need to complete the functions in usb20jtag_init, usb20jtag_quit and usb20jtag_execute_queue.

```
COMMAND_HANDLER(usb20jtag_handle_vid_pid_command)
{
    // TODO
    return ERROR_OK;
}

COMMAND_HANDLER(usb20jtag_handle_pin_command)
{
    // TODO
    return ERROR_OK;
}

static const struct command_registration usb20jtag_subcommand_handlers[] = {
    {
        .name = "vid_pid",
        .handler = usb20jtag_handle_vid_pid_command,
        .mode = COMMAND_CONFIG,
        .help = "",
        .usage = "",
    },
    {
        .name = "pin",
        .handler = usb20jtag_handle_pin_command,
        .mode = COMMAND_ANY,
        .help = "",
        .usage = "",
    },
    COMMAND_REGISTRATION_DONE
};

static const struct command_registration usb20jtag_command_handlers[] = {
    {
        .name = "usb20jtag",
        .mode = COMMAND_ANY,
        .help = "perform usb20jtag management",
        .chain = usb20jtag_subcommand_handlers,
        .usage = "",
    },
    COMMAND_REGISTRATION_DONE
};

static struct jtag_interface usb20jtag_interface = {
    .supported = DEBUG_CAP_TMS_SEQ,
    .execute_queue = usb20jtag_execute_queue,
};

struct adapter_driver usb20jtag_adapter_driver = {
    .name = "usb20jtag",
    .transports = jtag_only,
    .commands = usb20jtag_command_handlers,

    .init = usb20jtag_init,
    .quit = usb20jtag_quit,

    .jtag_ops = &usb20jtag_interface,
};
```

3.2.5. Construct API to call

According to the analysis of usb20jtag_adapter_driver, the API we need to build mainly include:

static int usb20jtag_init(void) // Device initialization function

static int usb20jtag_quit(void) // Device exit function

static intusb20jtag_execute_queue(void) // JTAG driver layer call function queue

The function corresponding to the code in usb20jtag_execute_queue is mainly for the functions such as timing switching and data transmission of the device.

```
static int usb20jtag_execute_queue(void)
{
    struct jtag_command *cmd;
    static int first_call = 1;
    int ret = ERROR_OK;

    if (first_call) {
        first_call--;
        USB20Jtag_Reset();
    }

    for (cmd = jtag_command_queue; ret == ERROR_OK && cmd;
         cmd = cmd->next) {
        switch (cmd->type) {
        case JTAG_RESET:
            USB20Jtag_Reset();
            break;
        case JTAG_RUNTEST:
            USB20JTAG_RunTest(cmd->cmd.runtest->num_cycles,
                        cmd->cmd.runtest->end_state);
            break;
        case JTAG_STABLECLOCKS:
            USB20JTAG_TableClocks(cmd->cmd.stableclocks->num_cycles);
            break;
        case JTAG_TLR_RESET:
            USB20Jtag_MoveState(cmd->cmd.statemove->end_state, 0);
            break;
        case JTAG_PATHMOVE:
            USB20Jtag_MovePath(cmd->cmd.pathmove);
            break;
        case JTAG_TMS:
            USB20Jtag_TMS(cmd->cmd.tms);
            break;
        case JTAG_SLEEP:
            USB20Jtag_Sleep(cmd->cmd.sleep->us);
            break;
        case JTAG_SCAN:
            ret = USB20JTAG_Scan(cmd->cmd.scan);
            break;
        default:
            LOG_ERROR("BUG: unknown JTAG command type 0x%X",
                  cmd->type);
            ret = ERROR_FAIL;
            break;
        }
    }
    return ret;
}
```

3.2.6. Parse part of the code

Take the functions of device initialization, reading/writing device, batch reading/writing, and state switching as examples. See notes for details.

usb20jtag_init

This function is used to initialize the device. First, USB2.0_JTAG uses a separate USB2.0 (480Mbps) high-speed device general driver. Therefore, first use the method of obtaining the library function address to initialize the operation function in the driver, and then open the device. Then set the transmission frequency, and call the *quit* function only after the *init* function is called successfully.

```
static int usb20jtag_init(void)
{
    if(hModule == 0)
    {
        hModule = LoadLibrary("CH375DLL.dll");
        if (hModule)
        {
            pOpenDev            = (pCH375OpenDevice)  GetProcAddress(hModule, "CH375OpenDevice");
            pCloseDev           = (pCH375CloseDevice) GetProcAddress(hModule, "CH375CloseDevice");
            pReadData           = (pCH375ReadData)  GetProcAddress(hModule, "CH375ReadData");
            pWriteData          = (pCH375WriteData) GetProcAddress(hModule, "CH375WriteData");
            pReadDataEndP       = (pCH375ReadEndP)  GetProcAddress(hModule, "CH375ReadEndP");
            pWriteDataEndP      = (pCH375WriteEndP) GetProcAddress(hModule, "CH375WriteEndP");
            pSetTimeout         = (pCH375SetTimeoutEx) GetProcAddress(hModule, "CH375SetTimeout");
            pSetBufUpload       = (pCH375SetBufUploadEx) GetProcAddress(hModule, "CH375SetBufUploadEx"
);
            pClearBufUpload     = (pCH375ClearBufUpload) GetProcAddress(hModule, "CH375ClearBufUpload"
);
            pQueryBufUploadEx   = (pCH375QueryBufUploadEx) GetProcAddress(hModule,
"CH375QueryBufUploadEx");
            pGetConfigDescr     = (pCH375GetConfigDescr) GetProcAddress(hModule, "CH375GetConfigDescr"
);
            if(pOpenDev == NULL || pCloseDev == NULL || pSetTimeout == NULL || pSetBufUpload == NULL ||
pClearBufUpload == NULL || pQueryBufUploadEx == NULL || pReadData == NULL || pWriteData == NULL ||
pReadDataEndP == NULL || pWriteDataEndP == NULL || pGetConfigDescr == NULL)
            {
                LOG_ERROR("GetProcAddress error ");
                return ERROR_FAIL;
            }
        }
        AfxDevIsOpened = pOpenDev(gIndex);
        if (AfxDevIsOpened == false)
        {
            gOpen = false;
            LOG_ERROR("USB20JTAG Open Error.");
            return ERROR_FAIL;
        }
        pSetTimeout(gIndex,1000,1000,1000,1000);
        USB20Jtag_ClockRateInit(0, 4);

        tap_set_state(TAP_RESET);
    }
    return 0;
}
```

The USB20Jtag_ClockRateInit function judges the connected interface at full speed or high speed. The lengths of the packets under different interfaces are different, as the single endpoint size of USB2.0 high-speed device is 512 bytes, and the size of the full-speed device is 64 bytes. As follows:

```
/**
 * USB20Jtag_ClockRateInit - 初始化USB20Jtag时钟速率
 * @param Index              USB20Jtag 设备操作句柄
 * @param iClockRate         设置的USB20Jtag时钟参数(0-5)
 *
 * iClockRate Value:
 * args:               0 - - - - 1 - - - - 2 - - - - 3 - - - - 4
 *                     |         |         |         |         |
 * clockeRate:       2.25MHz    4.5MHz     9MHz     18MHz     36MHz
 *
 */
static int USB20Jtag_ClockRateInit(unsigned long Index,unsigned char iClockRate)
{
    unsigned char mBuffer[256] = "";
    unsigned long mLength ,i,DescBufSize;
    bool  RetVal = false;
    unsigned char DescBuf[256] = "";
    unsigned char clearBuffer[8192] = "";
    unsigned long TxLen = 8192;
    PUSB_ENDPOINT_DESCRIPTOR       EndpDesc;
    PUSB_COMMON_DESCRIPTOR         UsbCommDesc;

    if( (iClockRate > 4) )
        goto Exit;

    // 获取的USB速度,默认为480MHz USB2.0高速，如果连接至全速HUB下则为12MHz USB全速。
    DescBufSize = sizeof(DescBuf);
    if( !pGetConfigDescr(gIndex,DescBuf,&DescBufSize))
        goto Exit;

    // 根据USB BULK端点大小来判断，如端点大小为512B，则为480MHz USB2.0高速
    AfxUsbHighDev = false;
    i = 0;
    while(i < DescBufSize)
    {
        UsbCommDesc = (PUSB_COMMON_DESCRIPTOR)&DescBuf[i];
        if( UsbCommDesc->bDescriptorType == USB_ENDPOINT_DESCRIPTOR_TYPE )
        {
            EndpDesc = (PUSB_ENDPOINT_DESCRIPTOR)&DescBuf[i];
            if( (EndpDesc->bmAttributes&0x03)==USB_ENDPOINT_TYPE_BULK )
            {
                if((EndpDesc->bEndpointAddress&USB_ENDPOINT_DIRECTION_MASK))
                {
                    DataUpEndp = EndpDesc->bEndpointAddress&(~USB_ENDPOINT_DIRECTION_MASK);
                    BulkInEndpMaxSize = EndpDesc->wMaxPacketSize;           // 端点大小
                    AfxUsbHighDev = (EndpDesc->wMaxPacketSize == 512);      // USB速度类型

                }
                else
                {
                    BulkOutEndpMaxSize = EndpDesc->wMaxPacketSize;
                    DataDnEndp = EndpDesc->bEndpointAddress;
                }
            }
        }
        i += UsbCommDesc->bLength;
    }
    // 根据USB速度,设置每个命令包最大数据长度
    if(AfxUsbHighDev)
    {
        USB_PACKET             =  USB_PACKET_USBHS;
        CMDPKT_DATA_MAX_BYTES  =  CMDPKT_DATA_MAX_BYTES_USBHS;              //507B
    }
    else
    {
        USB_PACKET             =  USB_PACKET_USBFS;
        CMDPKT_DATA_MAX_BYTES  =  CMDPKT_DATA_MAX_BYTES_USBFS;              //59B
    }
    CMDPKT_DATA_MAX_BITS = CMDPKT_DATA_MAX_BYTES/16*16/2;                   //每个命令所传输的最大位数，每位需由两个字节表示，取2字节的整数倍

    // 根据硬件缓冲区大小计算每批量传输传输的位数,多命令拼包
    MaxBitsPerBulk = HW_TDO_BUF_SIZE/CMDPKT_DATA_MAX_BYTES*CMDPKT_DATA_MAX_BITS;
    // 根据硬件缓冲区大小计算每批量传输传输的字数,多命令拼包
    MaxBytesPerBulk = HW_TDO_BUF_SIZE - (HW_TDO_BUF_SIZE+CMDPKT_DATA_MAX_BYTES-1)/CMDPKT_DATA_MAX_BYTES*3;;

    // USB BULKIN上传数据采用驱动缓冲上传方式,较直接上传效率更高
    pSetBufUpload(gIndex, true, DataUpEndp, 4096);                          //指上传端点为缓冲上传模式，缓冲区大小为8192
    pClearBufUpload(gIndex, DataUpEndp);                                    //清空驱动内缓冲区数据

    if (!USB20Jtag_Read(clearBuffer, &TxLen))                               //读取硬件缓冲区数据
    {
        LOG_ERROR("USB20Jtag_WriteRead read usb data failure.");
        return 0;
    }

    //构建USB JTAG初始化命令包，并执行
    i = 0;
    mBuffer[i++] = USB20Jtag_CMD_JTAG_INIT;
    mBuffer[i++] = 6;
    mBuffer[i++] = 0;
    mBuffer[i++] = 0;                                                       //保留字节
    mBuffer[i++] = iClockRate;                                              //JTAG时钟速度
    i += 4;                                                                 //保留字节
    mLength = i;
    if( !USB20Jtag_Write(mBuffer,&mLength) || (mLength!=i) )
        goto Exit;

    // 读取返回值并判断初始化是否成功
    mLength = 4;
    if( !USB20Jtag_Read(mBuffer,&mLength) || (mLength!=4) )
    {
        LOG_ERROR("USB20Jtag clock initialization failed.\n");
        goto Exit;
    }
    RetVal = ( (mBuffer[0] == USB20Jtag_CMD_JTAG_INIT) && (mBuffer[USB20Jtag_CMD_HEADER]==0) );
Exit:
    return (RetVal);
}
```

USB20Jtag_Write

```
/**
 *  USB20Jtag_Write - USB20Jtag 写方法
 *  @param oBuffer    指向一个缓冲区,放置准备写出的数据
 *  @param ioLength   指向长度单元,输入时为准备写出的长度,返回后为实际写出的长度
 *
 *  @return           写成功返回1，失败返回0
 */
static int USB20Jtag_Write(void* oBuffer,unsigned long* ioLength)
{
    unsigned long wlength = *ioLength;
    int ret = pWriteData(gIndex, oBuffer, ioLength);
    LOG_DEBUG_IO("(size=%d, DataDnEndp=%d, buf=[%s]) -> %" PRIu32, wlength, DataDnEndp, HexToString((
uint8_t*)oBuffer, *ioLength),
                *ioLength);
    return ret;
}
```

USB20Jtag_Read

```
/**
 * USB20Jtag_Read - USB20Jtag 读方法
 * @param oBuffer   指向一个足够大的缓冲区,用于保存读取的数据
 * @param ioLength  指向长度单元,输入时为准备读取的长度,返回后为实际读取的长度
 *
 * @return          读成功返回1，失败返回0
 */
static int USB20Jtag_Read(void* oBuffer,unsigned long* ioLength)
{
    unsigned long rlength = *ioLength, packetNum, bufferNum, RI, RLen, WaitT = 0, timeout = 20;
    int ret = false;

    // 单次读取最大允许读取4096B数据，超过则按4096B进行计算
    if (rlength > HW_TDO_BUF_SIZE)
        rlength = HW_TDO_BUF_SIZE;

    RI = 0;
    while (1)
    {
        RLen = 8192;
        if ( !pQueryBufUploadEx(gIndex, DataUpEndp, &packetNum, &bufferNum))
            break;

        if (!pReadDataEndP(gIndex, DataUpEndp, oBuffer+RI, &RLen))
        {
            LOG_ERROR("USB20Jtag_Read read data failure.");
            goto Exit;
        }
        RI += RLen;
        if (RI >= *ioLength)
            break;
        if (WaitT++ >= timeout)
            break;
        Sleep(1);
    }
    LOG_DEBUG_IO("(size=%d, DataDnEndp=%d, buf=[%s]) -> %" PRIu32, rlength, DataUpEndp,HexToString((
uint8_t*)oBuffer, *ioLength),*ioLength);
    ret = true;
Exit:
    *ioLength = RI;
    return ret;
}
```

USB20Jtag_TmsChange

This function is used for JTAG TAP state switching. TMS controls which shift register is placed on the device between TDI and TDO. The figure below shows the state change accompanying the TMS change.

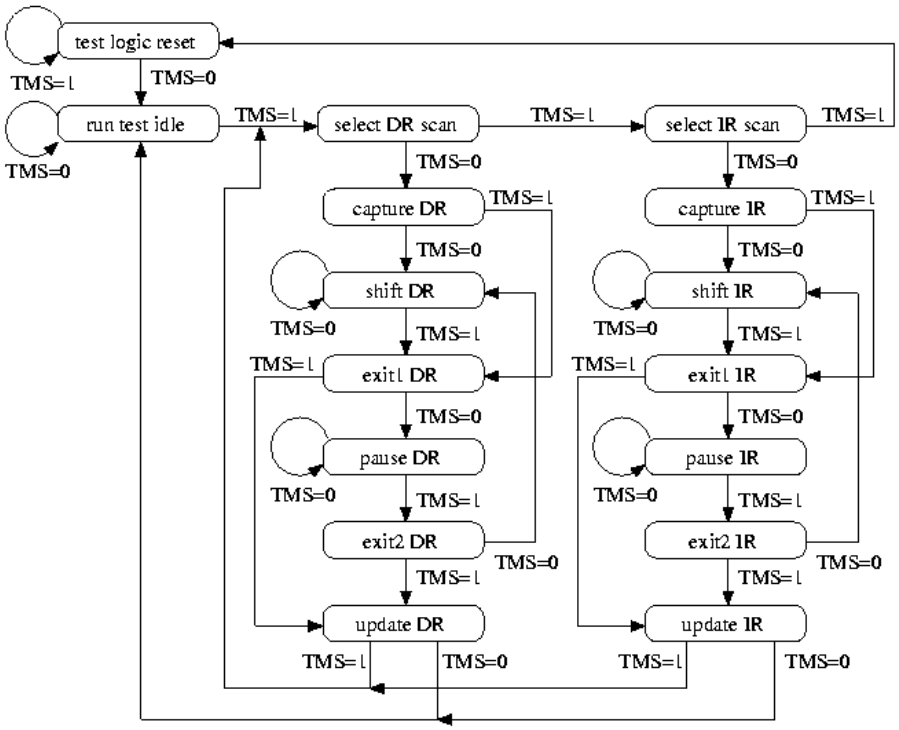

Figure 6 JTAG Test Access Port (TAP) controller state transition diagram

This function combines the TMS value from a TAP state to a TAP state into an 8-bit data. By passing in the step and skip parameters, it is judged from the TMS value of the skip bit to the end of the step value, which can implement state switching. Combined with the USB2.0_JTAG protocol built by CH32V305/CH32V307, the following code can be completed:

```
/**
 * USB20Jtag_TmsChange - 功能函数，通过改变TMS的值来进行状态切换
 * @param tmsValue       需要进行切换的TMS值按切换顺序组成一字节数据
 * @param step           需要读取tmsValue值的位值数
 * @param skip           从tmsValue的skip位处开始计数到step
 *
 */
static void USB20Jtag_TmsChange(const unsigned char* tmsValue, int step, int skip)
{
    int i;
    unsigned long BI,retlen,TxLen;
    unsigned char BitBangPkt[4096] = "";

    BI = USB20Jtag_CMD_HEADER;
    retlen = USB20Jtag_CMD_HEADER;
    LOG_DEBUG_IO("(TMS Value: %02x..., step = %d, skip = %d)", tmsValue[0], step, skip
);

    for (i = skip; i < step; i++)
    {
        retlen = USB20Jtag_ClockTms(BitBangPkt,(tmsValue[i/8] >> (i % 8)) & 0x01, BI);
        BI = retlen;
    }
    retlen = USB20Jtag_IdleClock(BitBangPkt, BI);
    BI = retlen;

    BitBangPkt[0] = USB20Jtag_CMD_JTAG_BIT_OP;
    BitBangPkt[1] = (unsigned char)BI - USB20Jtag_CMD_HEADER;
    BitBangPkt[2] = 0;

    TxLen = BI;

    if (!USB20Jtag_Write(BitBangPkt, &TxLen) && (TxLen != BI))
    {
        LOG_ERROR("JTAG Write send usb data failure.");
        return NULL;
    }
}
```

3.3. Compile OpenOCD that supports USB2.0_JTAG

3.3.1. Build Cygwin compilation environment

1. Get the Cygwin installation packages from: http://www.cygwin.com/install.html

2. Install Cygwin. You can choose the required tools for installation. For the installation tools, please refer to the following (Here select the latest version supported by Cygwin)
   autobuild, autoconf, autogen, automake, automoc4, bison, clang, cmake cygwin32-libtool, dos2unix, doxygen, gcc-core, gcc-g++, gdb, git, libc++-devel, libftdi1, libftdi1-devel, libhidapi-devel, libhidapi0, libtool, libusb-devel, libusb1.0, make, meson, mingw64-i686-libusb1.0, mingw64-x86_64-libusb, mingw64-x86_64-libusb1.0, pkg-config, usbutils, wget, wget2.

3. Enter the modified *openocd* folder, and run the following command.

```
./bootstrap
```

```
$ ./bootstrap
+ aclocal --warnings=all
+ libtoolize --automake --copy
+ autoconf --warnings=all
+ autoheader --warnings=all
+ automake --warnings=all --gnu --add-missing --copy
Setting up submodules
Submodule path 'jimtcl': checked out 'a77ef1a6218fad4c928ddbdc03c1aedc41007e70'
Generating build system...
configure.ac:38: warning: The macro `AC_PROG_CC_C99' is obsolete.
configure.ac:38: You should run autoupdate.
/mnt/share/cygpkgs/autoconf2.7/autoconf2.7.noarch/src/autoconf-2.71/lib/autoconf/c.m4:1659: AC_PROG_CC_C99 is expanded from...
configure.ac:38: the top level
Bootstrap complete. Quick build instructions:
./configure ....
```

```
// If other debuggers are needed to be disabled,  add them after the configure command.
Openocd is enabled by default.
./configure --enable-usb20jtag --host=i686-w64-mingw32CFLAGS='-g -O0'
```

After that, it can be seen that USB2.0_JTAG is supported in the OpenOCD configuration summary.

```
libjaylink configuration summary:
 - Package version ................. 0.2.0
 - Library version ................. 1:0:1
 - Installation prefix ............. /usr/local
 - Building on ..................... x86_64-pc-cygwin
 - Building for .................... i686-w64-mingw32

Enabled transports:
 - USB ............................. yes
 - TCP ............................. yes



OpenOCD configuration summary
--------------------------------------------------
MPSSE mode of FTDI based devices        yes (auto)
ST-Link Programmer                      yes (auto)
TI ICDI JTAG Programmer                 yes (auto)
Keil ULINK JTAG Programmer              yes (auto)
USB20JTAG Programmer                    yes
Altera USB-Blaster II Compatible        yes (auto)
Bitbang mode of FT232R based devices    yes (auto)
Versaloon-Link JTAG Programmer          yes (auto)
TI XDS110 Debug Probe                   yes (auto)
CMSIS-DAP v2 Compliant Debugger         yes (auto)
OSBDM (JTAG only) Programmer            yes (auto)
eStick/opendous JTAG Programmer         yes (auto)
Olimex ARM-JTAG-EW Programmer           yes (auto)
Raisonance RLink JTAG Programmer        yes (auto)
USBProg JTAG Programmer                 yes (auto)
Andes JTAG Programmer                   yes (auto)
CMSIS-DAP Compliant Debugger            no
Nu-Link Programmer                      no
Cypress KitProg Programmer              no
Altera USB-Blaster Compatible           no
ASIX Presto Adapter                     no
OpenJTAG Adapter                        no
Linux GPIO bitbang through libgpiod     no
SEGGER J-Link Programmer                yes (auto)
Bus Pirate                              no
Use Capstone disassembly framework      no
```

Finally, run "make install" to compile and install. The compiled *OpenOCD.exe* file appears in the *openocd/src* folder. If it is needed to be output to a specified directory, you can add --prefix=xxxx (path name) in ./configure to specify. In this case, the entire JTAG debugger solution based on CH32V305/CH32V307 is completed.

4. Summary

This solution is to implement a USB2.0 (480Mbps) to JTAG debugger (USB2.0_JTAG) based on CH32V305/CH32V307. Combined with OpenOCD with the added USB2.0_JTAG interface, it can implement the JTAG debugging and downloading functions. The related resources are available, including reference schematic diagram, MCU program source code, USB2.0 (480Mbps) high-speed device general driver, USB to JTAG/SPI function library source code, reference program source code, communication protocol, etc., engineers can modify and develop as required for more application scenarios.