

高速 USB2.0 (480Mbps) JTAG 调试器单芯片方案

一、前言

基于 CH32V305/CH32V307 系列 MCU 实现的 USB2.0 (480Mbps) 转 JTAG 接口方案, 可用于调试或下载 CPU、DSP、FPGA 和 CPLD 等器件。方案构成上只需一颗 CH32V305/307 芯片, 无需 CPLD 和 USB PHY 辅助芯片。

当前方案中构建的 JTAG 调试编程器命名为 USB2.0_JTAG, 使用自定义 BitBang 和 ByteShift 协议传输模式, 其中 JTAG 写速度可达 36Mbit/s, 上位机软件则采用 OpenOCD, 通过在 OpenOCD 中添加属于 USB2.0_JTAG 的接口后, OpenOCD 将支持操作 USB2.0_JTAG 来实现 JTAG 下载调试功能, 此次方案中下载采用的目标 FPGA 是 XILINX SPARTAN-6 XC6SLX9, 其下载指令如下:

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

其中下载文件与下载完成图例如下 (下载文件 led.bit 大小为 333KB):

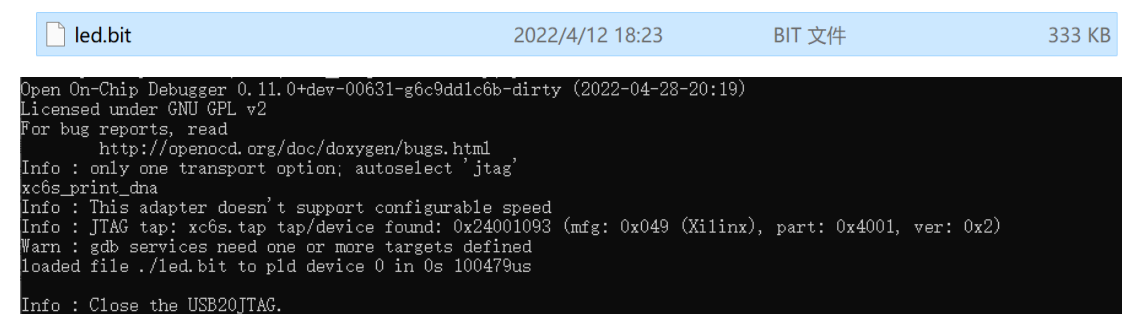






图 1 USB2.0_Jtag 下载

与市面上的下载调试器进行下载对比:

Jtag 调试器	实物图片	下载文件大小	下载时间	下载速度
USB2.0_Jtag		333KB	0.100s	3.33MB/s
XXX USB-Bxxxxx		333KB	1.709s	194.88KB/s

XXX USB-Bxxxxx		333KB	1.166s	285.59KB/s
XXX USB-Bxxxxx II		333KB	0.178s	1.83MB/s

方案结构框图如下所示：

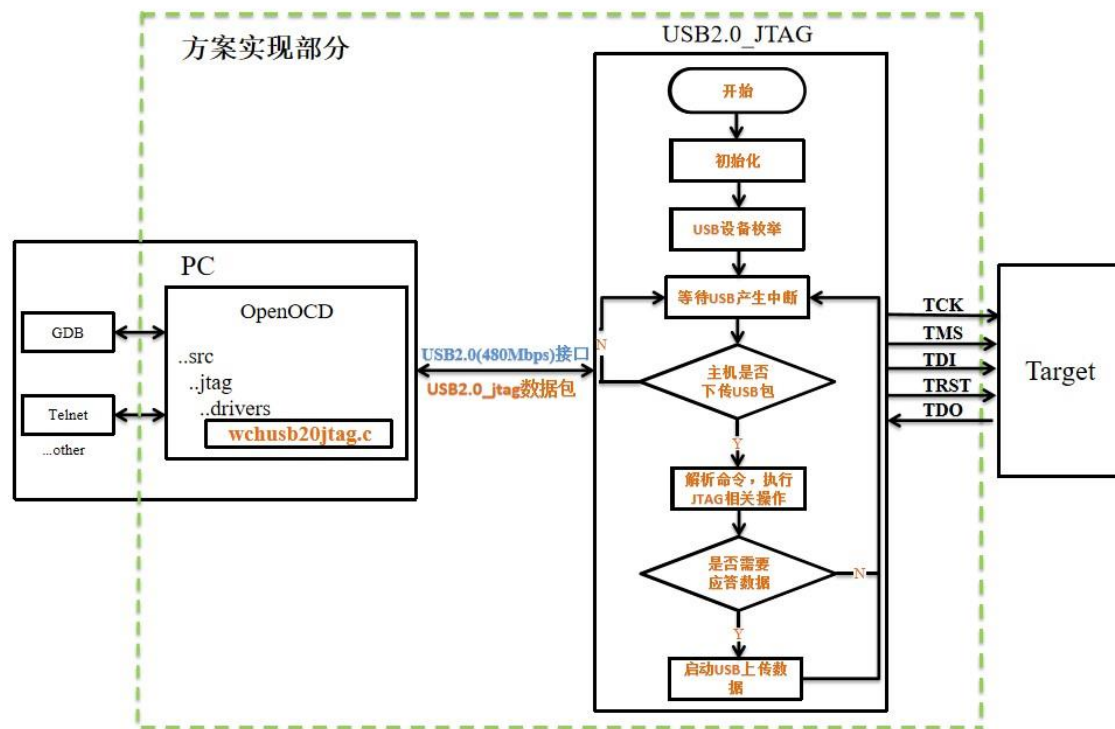


图2 CH32V305/CH32V307 构建 JTAG 调试器方案框图

本方案将提供完整的开发资料，包括参考原理图、MCU 程序源码、USB2.0 (480Mbps) 高速设备通用驱动、USB 转 JTAG/SPI 函数库源码、例子程序源码、通信协议。

实现本方案所需的软硬件环境如下：

硬件：CH32V305/307 开发板

软件：OpenOCD 源码

编译工具：Cygwin（此处编译工具的安装参考 3.2.1 小节内容）

FPGA 目标板：XILINX SPARTAN-6 XC6SLX9

芯片相关资料下载链接:

MCU IDE (MounRiver): <http://www.mounriver.com/download>

MCU ISP Tool: http://www.wch.cn/downloads/WCHISPTool_Setup_exe.html

USB 驱动程序: http://www.wch.cn/downloads/CH372DRV_EXE.html

二、硬件方案实现

本方案使用 CH32V305/307 开发板进行测试, CH32V305/307 系列芯片是基于青稞 32 位 RISC-V 设计的工业级通用微控制器。配备了硬件堆栈区、快速中断入口, 在标准 RISC-V 基础上大大提高了中断响应速度。搭载 V4F 内核, 主频支持 144MHz, 独立 GPIO 供电, 内置 2 个 12 位 ADC 模块、2 个 12 位 DAC 模块、多组定时器、多通道触摸按键电容检测 (TKEY) 等功能, 还包含标准和专用通讯接口: I2C、I2S、SPI、USART、SDIO、CAN 控制器、USB2.0 全速主机/设备控制器、USB2.0 (480Mbps) 高速主机/设备控制器 (内置自研 PHY 收发器, 无需向第三方支持 IP 费用, 故成本与全速 USB 相当)、数字图像接口、千兆以太网控制器等。144MHz 的系统主频, 具有更高的运算能力和更快的处理速度, 64 脚及以上封装芯片还支持 GPIO 独立供电 (支持 1.8、2.5 和 3.3V)。另外, USB 接口和 SPI 接口均具有硬件 DMA, 这也是本方案速度较快的关键之处。

CH32V305/307 系列芯片相关资料可通过访问官网 <http://wch.cn> 获取。

使用引脚定义如下:

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

2.1、 固件结构

2.1.1、 USB 数据下传

CH32V305/307 芯片在初始化时, 启用高速 USB2.0 (480Mbps) 接口, 端点大小设置为 512 字节, 且为提高下传及处理效率, 为 USB 下传分配了 4096 字节的环形缓冲区, 以确保在处理前面下传数据的同时, 还能继续接收后续 USB 数据。

具体下传流程为:

步骤 1: 计算机下传一包 USB 数据, MCU 产生端点 OUT 成功中断;

步骤 2: MCU 切换下传端点的下一次接收 DMA 地址 (仅切换 DMA 地址, 不需要进行数据拷贝), 并计算相关变量;

步骤 3: MCU 判断是否还有足够缓冲区接收下一包数据, 如果缓冲区足够则设置端点允许继续接收数据; 如果缓冲区不足则设置端点暂停接收数据, 只有前面数据处理后有足够缓冲区再次设置端点允许继续接收数据。

步骤 4: 继续重复 1-3 步骤;

2.1.2、 命令解析处理

JTAG 接口一共有 5 条命令, 如下表所示:

0xD0	JTAG 接口初始化命令 (DEF_CMD_JTAG_INIT)
0xD1	JTAG 接口引脚位控制命令 (DEF_CMD_JTAG_BIT_OP)
0xD2	JTAG 接口引脚位控制并读取命令 (DEF_CMD_JTAG_BIT_OP_RD)
0xD3	JTAG 接口数据移位命令 (DEF_CMD_JTAG_DATA_SHIFT)
0xD4	JTAG 接口数据移位并读取命令 (DEF_CMD_JTAG_DATA_SHIFT_RD)

命令格式为：

CMD	LEN	DATA
命令码	后续数据长度	后续数据
1 个字节	2 个字节	N 个字节 (0<=N<=507)

以下为命令的大致介绍：

- 1) D0 命令用于初始化 JTAG 的模式和速度。
- 2) D1 命令用于控制 TCK、TMS、TDI 和 TRST 引脚，1 个字节的数据可以控制四个引脚输出不同的电平，主要用于 JTAG 状态机的切换。
- 3) D2 命令同样用于控制 TCK、TMS、TDI 和 TRST 引脚，并且在控制 4 个引脚输出电平的同时读取 TD0 引脚上的数据。D2 命令主要用于状态机在切换到 SHIFT-I (D)R 进行 IR 或者 DR 的写操作时，同时读取目标设备从 TD0 返回的数据。
- 4) D3 命令用于传输 TDI 上需要发送的数据。在使用 D3 命令时，TMS 和 TRST 引脚无法控制，CH32V305 控制产生 8 个周期的 TCK，TDI 来自 D3 命令传输的字节数据。D3 命令主要用于在 SHIFT-I (D)R 状态时，通过 TDI 批量写 IR 或者 DR。
- 5) D4 命令与 D3 命令的功能基本相同，区别是在批量传输数据的同时读取目标设备从 TD0 引脚上返回的数据。

由于在写 IR 或者写 DR 时最后一位数据是在进入 Exit1-I (D)R 状态的那个上升沿捕获的，因此如果要向 I (D)R 写 N 位数据，在 SHIFT-I (D)R 状态时可以通过 D3 (D4) 命令写 (N/8) 个字节数据，剩余 (N%8 - 1) 位数据使用 D1 (D2) 命令写入，最后一位数据通过 D1 (D2) 命令切换到 Exit1-I (D)R 状态的同时写入 I (D)R。

为提高命令下传及处理效率，计算机可以将多条命令合并到一个 USB 包下传或者多个连续的 USB 包下传，MCU 可自动进行分包。

具体处理流程为：

步骤 1：MCU 判断是否有剩余 USB 下传数据未处理，如果没有则继续等待，如果有则取出一包数据 (单包最多为 512 字节) 进行处理；

步骤 2：MCU 对取出的数据包依次进行分析处理，并执行相对操作；如果有回读数据则存储在对应的 JTAG 接收缓冲区中。支持单个 USB 数据包包含多条命令，也支持单条命令跨 2 个 USB 包

步骤 3：MCU 切换缓冲区处理指针并计算相关变量；

步骤 4：继续重复 1-3 步骤；

2.1.3、USB 数据上传

执行 D2 或 D4 命令时，会从目标设备中读取返回数据，该数据存储在接收缓冲区中。接收缓冲区为 4096 字节的环形缓冲区，以便在存储数据的同时，不影响数据的上传。

具体上传流程为：

步骤 1：MCU 在执行 D2 或 D4 命令时，回读目标设备返回的数据，并存储在接收缓冲区中；

步骤 2: MCU 执行完本条命令之后, 判断是否有回读数据需要上传, 如果没有数据需要上传, 则继续执行下一条命令; 如果有回读数据需要上传, 则设置端点上传 DMA 地址, 并启动 USB 数据上传(仅需设置 DMA 地址, 不需要进行数据拷贝);

步骤 3: MCU 等待数据是否上传成功, 如果上传成功则计算相关变量, 否则等待超时后取消本次数据上传, 以便下次继续上传剩余数据;

步骤 4: 继续重复 1-3 步骤;

2.2、 部分代码解析

本节主要介绍一下 D0-D4 命令的代码实现。

(1)、D1(JTAG 接口引脚位控制命令):

```
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 字节数据可以通过设置引脚对应 bit 来控制 TDI、TMS、TCK、TRST 四个引脚的电平, 两个字节可以完成一个时钟周期。例如 0x13 代表 TDI、TMS、TCK 引脚都为高, TRST 引脚设置为低。

(2)、D2(JTAG 接口引脚位控制并读取命令):

D2 命令同样使用 JTAG_Port_BitShift 函数, 区别在于在判断下传数据对应的 TCK 引脚为高时, 会读取 TDO 引脚电平并放入上传缓冲区内。

(3)、D3(JTAG 接口数据移位命令):

```
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 );
    }
}
```

D4 命令采用手动设置 TCK 的方式，将每一字节数据拆分为 8bit，然后通过拉高和拉低 TCK 引脚完成 8 个时钟周期，发送 TDI 并读取 TDO 引脚电平。

三、软件方案实现

3.1、OpenOCD 简介

OpenOCD (Open On-Chip Debugger) 开放片上调试器是一个免费开源项目，旨在为嵌入式目标设备提供调试、系统内编程和边界扫描测试。结合调试适配器将完成以上功能，调试适配器是一个小型硬件模块，为其需调试的目标硬件提供所需的电信号，此方案就是使用 CH32V305/CH32V307 实现一个调试器。打开 openocd 源码目录，其主要内容集中在 src 目录下，flash 目录下是对各种芯片内置 FLASH 等读写擦除等操作的实现；helper 目录下是命令行、command 分析的实现，rtos 是 openocd 对操作系统支持的实现，server 目录是 GDB TELNET 等远程连接服务的实现，target 目录是 openocd 支持的各种目标架构的实现，jtag 目录下是传输协议层的实现，此方案我们只需要在 jtag/drivers 目录下，实现一个新的调试器接口即可。

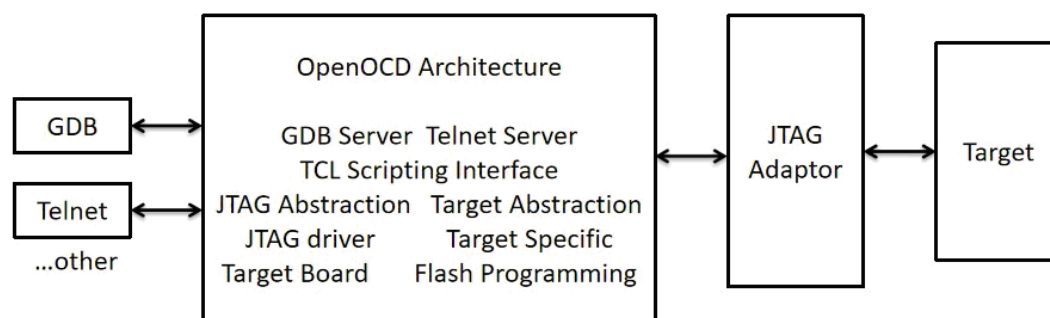


图 3 OpenOCD 结构框图

3.2、在 OpenOCD 中添加 USB2.0_JTAG 接口支持

3.2.1、Cygwin 编译环境搭建

- 1、此方案使用 Cygwin 对 openocd 源码进行编译，可从官网获取其 Cygwin 安装程序
下载链接: [Cygwin Installation](#)
- 2、安装 Cygwin，请参考如下表格选择对应 Package 与 Version 进行安装：

Package	Version	Package	Version
autobuild	5.3-1	bison	3.8.2-1
autoconf	15-1	clang	8.0.1-1
autogen	5.18.16-1	cmake	3.20.0-1
automake	11-1	cygwin32-libtool	2.4.6-3
automoc4	0.9.88-10	libtool	2.4.7-1
automake1.4	1.4p6-11	automake1.6	1.6.3-12
automake1.5	1.5-11	automake1.7	1.7.9-11
automake1.8	1.8.5-11	gdb	10.2-1
dos2unix	7.4.2-1	git	2.36.0-1
doxygen	1.9.2-2	libc++-devel	8.0.1-1
gcc-core	11.2.0-1	libftdi1	1.4-2

gcc-g++	11.2.0-1	libftdi1-devel	1.4-2
libhidapi-devel	0.8.0-rc1-1	libtool	2.4.7-1
libhidapi0	0.8.0-rc1-1	libusb1.0	1.0.21-1
libusb-devel	1.2.6.0-2	meson	0.60.3-1
make	4.3-1	mingw64-x86_64-libusb	1.2.6.0-1
mingw64-i686-libusb1.0	1.0.21-1	pkg-config	1.8.0-1
mingw64-x86_64-libusb1.0	1.0.21-1	wget	1.21.3-1
usbutils	007-1	wget2	2.0.0-1

3、等待安装结束即可。

3.2.2、在 OpenOCD 中新建 USB2.0_JTAG 接口

1、打开 Cygwin，输入 `git clone git://git.code.sf.net/p/openocd/code openocd-code` 获取 OpenOCD 源码到文件夹 `openocd-code` 中；

2、在 Cygwin 命令行输入 `cd openocd-code` 进入到获取到 OpenOCD 源码目录下，修改该目录下 `configure.ac` 文件，添加对 USB2.0_JTAG 的驱动支持，修改如下（因 OpenOCD 为开源项目，故每次获取到的源码可能有所新增或删减，行数可能不一致，注意观察代码块格式添加即可）：

```
+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、修改工程目录下 `src/jtag/interfaces.c`，添加 `usb20jtag_adapter_driver` 调试器，修改如下：

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

4、修改工程 `/src/jtag/drivers` 目录下 `Makefile.am`，添加编译支持，修改如下：

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


3.2.3、 构建 USB2.0_JTAG 接口代码

在 jtag/drivers 目录下, 添加 usb20jtag.c 文件, 要构建 USB2.0_JTAG 的接口代码可从如下几个方面入手 (参考 3.2.3-7 小结)。也可将方案所提供的 usb20jtag.c 放入该目录下, 然后直接查看 3.3 节内容进行编译操作。

3.2.4、 USB2.0_JTAG 协议机制

使用 CH32V305 构建的 USB2.0_JTAG 在 BitBang 和 Byteshift 模式的基础上增添了包头命令码与发包长度设置, 其命令码格式与协议传输格式参考 2.1.2 小节。

3.2.5、 接口入口 API

在 3.2.2 中完成了对 usb20jtag_adapter_driver 调试器结构体进行了注册, 其内部结构如下:

```
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  
};
```

其中对应的接口函数 usb20jtag_interface 如下所示:

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

OpenOCD 中的驱动层通过对 usb20jtag_adapter_driver 的调用使其能对设备进行初始化和退出操作, 在接口函数 usb20jtag_interface 中则定义有被 JTAG 协议层所调用的执行命令队列 usb20jtag_execute_queue, 通过 JTAG 协议层下发的 CMD 数据则完成对应的操作, 因此我们只需完成 usb20jtag_init、usb20jtag_quit 以及 usb20jtag_execute_queue 中函数即可。

```
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,
};
```

3.2.6、构造 API 进行调用

根据对 usb20jtag_adapter_driver 的分析得出，我们需要构建的 API 主要包括

```
static int usb20jtag_init(void)           // 设备初始化函数
```

```
static int usb20jtag_quit(void)           // 设备退出函数
static intusb20jtag_execute_queue(void)    // JTAG 驱动层调用函数队列
```

其中 usb20jtag_execute_queue 内代码所对应的函数主要为实现设备完成时序切换、数据传输等功能。

```
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;
}
```

3.2.7、部分代码解析

以初始化设备、设备读写、批量读写、状态切换函数为例，请看下列代码中的中文注释。

usb20jtag_init

该函数用于初始化设备，首先 USB2.0_JTAG 使用的为单独的 USB2.0 (480Mbps) 高速设备通用驱动，故先使用获取库函数地址方式对驱动中的操作函数进行初始化，其次对设备进行打开操作后进行传输频率设置，仅有当 init 函数调用成功之后才会调用 quit 函数。

```
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;
}
```

其中涉及到的 USB20Jtag_ClockRateInit 函数中对所连接的接口进行了全速和高速的判断，不同接口下的组包长度将会不同，因为 USB2.0 高速设备单端点大小为 512Byte，全速则为 64Byte，其详细实现如下：

```

/**
 * 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
 *                |         |         |         |         |
 * ClockRate:     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;
    USB_ENDPOINT_DESCRIPTOR EndpDesc;
    USB_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 = (USB_COMMON_DESCRIPTOR)&DescBuf[i];
        if( UsbCommDesc->bDescriptorType == USB_ENDPOINT_DESCRIPTOR_TYPE )
        {
            EndpDesc = (USB_ENDPOINT_DESCRIPTOR)&DescBuf[i];
            if( (EndpDesc->bAttributes&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;
        CNDPKT_DATA_MAX_BYTES = CNDPKT_DATA_MAX_BYTES_USBHS; // 507B
    }
    else
    {
        USB_PACKET = USB_PACKET_USBFS;
        CNDPKT_DATA_MAX_BYTES = CNDPKT_DATA_MAX_BYTES_USBFS; // 59B
    }
    CNDPKT_DATA_MAX_BITS = CNDPKT_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, 多命令打包
    MaxBitsPerBulk = HW_TDO_BUF_SIZE/CNDPKT_DATA_MAX_BYTES*CNDPKT_DATA_MAX_BITS;
    // 根据硬件缓冲区大小计算每批量传输传输的字数, 多命令打包
    MaxBytesPerBulk = HW_TDO_BUF_SIZE - (HW_TDO_BUF_SIZE+CNDPKT_DATA_MAX_BYTES-1)/CNDPKT_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;
    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

该函数用于 JTAG TAP 状态的切换, 其中 TMS 控制在 TDI 和 TDO 之间的设备上放置哪个移位寄存器, 下图显示了 TMS 变化伴随的状态改变。

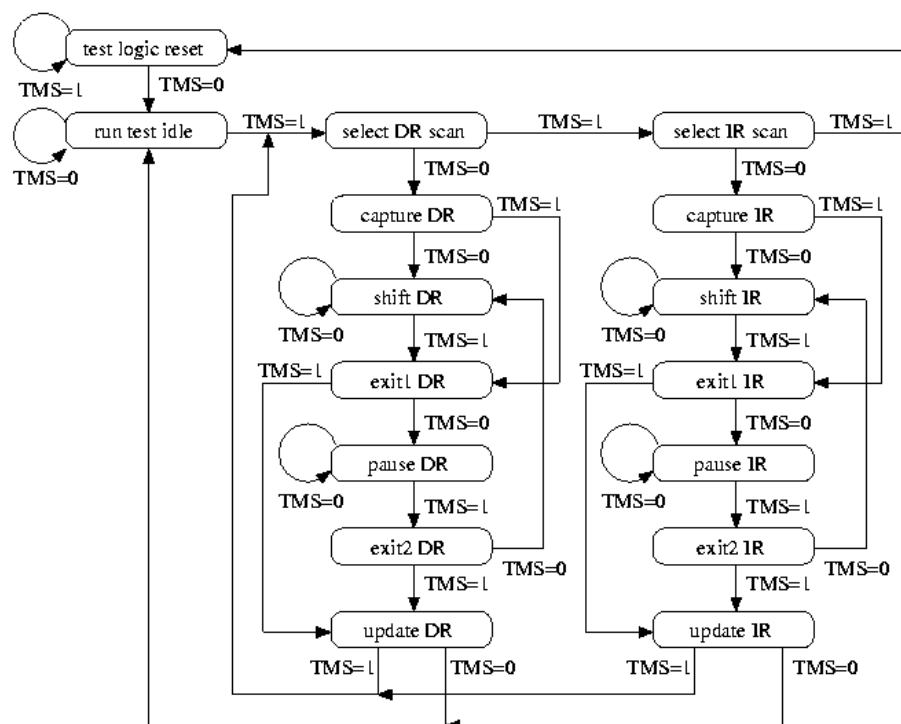


图 6 JTAG Test Access Port(TAP)controller state transition diagram

该函数将会把从 TAP 某状态到 TAP 某状态的 TMS 值组合成 8bit 数据, 通过传入参数 step 和 skip 来判断从第 skip 位 TMS 值开始到 step 值结束, 这将完成状态切换, 结合 CH32V305/CH32V307 构建的 USB2.0_JTAG 协议, 则可完成如下代码:


```

/**
 * 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、编译支持 USB2.0_JTAG 的 OpenOCD

1、进入已修改好的 OpenOCD 源码文件夹 openocd-code 下，运行如下命令

```
./bootstrap
```

当出现如下图所示内容代表第一步完成；

```

$ ./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 ....

```

2、若只使用自己构建的 JTAG 调试器接口，可运行如下所示配置命令，其中/home/Test/deploy 为编译出的 OpenOCD 可执行文件以及相关附属文件的存放目录，若需要输出到其他指定目录，

可在./configure 中修改--prefix=xxxx(路径名称)进行指定。

```
./configure --prefix=/home/Test/deploy --disable-doxygen-html
--disable-doxygen-pdf --disable-gccwarnings --disable-wextra --disable-stlink
--disable-ti-icdi --disable-ulink --disable-usb-blaster-2 --disable-ft232r
--disable-vsllink --disable-xds110 --disable-osbdm --disable-opensw
--disable-jlink --disable-aice --disable-ftdi --disable-internal-libjaylink
--disable-jtag_vpi --disable-cmsis-dap --disable-cmsis-dap-v2
--disable-usb-blaster --disable-usbprog --disable-rlink --disable-armjtagew
--disable-openjtag --disable-presto --enable-usb20jtag --host=i686-w64-mingw32
CFLAGS='-g -O0'
```

若需支持其他调试器，则可使用如下指令，但该条命令因各自系统环境不同可能需要额外安装其他环境依赖文件，需自己根据实际情况进行调整修改，OpenOCD 默认全部接口都支持，需要禁用其他调试接口的请参照上方指令。

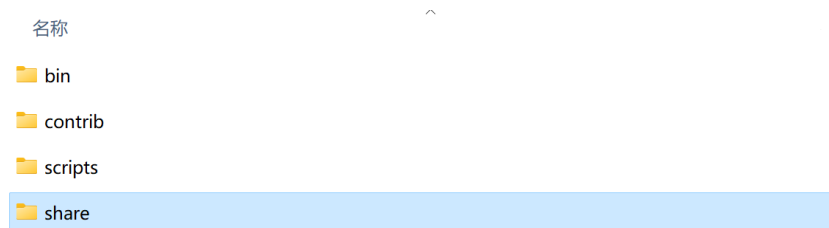
```
./configure --enable-usb20jtag --host=i686-w64-mingw32CFLAGS='-g -O0'
```

本方案中选择使用第一条指令，即只编译 USB20Jtag 调试器接口。

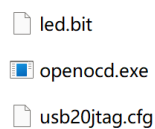
执行完这一步，可见 OpenOCD configuration summary 已出现对 USB2.0_JTAG 的支持

```
OpenOCD configuration summary
-----
MPSSE mode of FTDI based devices      no
ST-Link Programmer                    no
TI ICDI JTAG Programmer                no
Keil ULINK JTAG Programmer             no
USB20JTAG Programmer                  yes
Altera USB-Blaster II Compatible       no
Bitbang mode of FT232R based devices  no
Versaloon-Link JTAG Programmer         no
TI XDS110 Debug Probe                 no
CMSIS-DAP v2 Compliant Debugger        no
OSBDM (JTAG only) Programmer           no
eStick/opensw JTAG Programmer          no
Olimex ARM-JTAG-EW Programmer          no
Raisonance RLink JTAG Programmer       no
USBProg JTAG Programmer                no
Andes JTAG Programmer                  no
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 libgpod     no
SEGGER J-Link Programmer               no
Bus Pirate                            no
Use Capstone disassembly framework     no
```

最后在当前路径下运行“make install”进行编译安装，编译生成的 OpenOCD.exe 文件等将会出现在/home/Test/deploy 文件夹下，若需要实现方案开头的下载效果，还需将/home/Test/deploy 目录下 share/openocd 文件夹中“contrib”与“scripts”文件夹拷贝至 bin 目录同级，文件夹最后调整成如下图：



进入 bin 文件夹中，添加下载所需的“xxx.bit”文件与 USB20Jtag 配置文件“usb20jtag.cfg”，如下图所示：



使用 Windows 命令行工具 CMD 进入 bin 目录下，连接好设备后执行如下命令即可实现 .bit 文件的下载。

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

此时整个基于 CH32V305/CH32V307 构建的 JTAG 调试器方案就已完成。

四、总结

本方案基于 CH32V305/CH32V307 MCU 实现的 USB2.0(480Mbps) 转 JTAG 调试器 USB2.0_JTAG，结合添加 USB2.0_JTAG 接口的 OpenOCD 来完成 JTAG 调试下载功能，方案提供完整的资料，包括参考原理图、MCU 程序源码、USB2.0(480Mbps) 高速设备通用驱动、USB 转 JTAG/SPI 函数库源码、例子程序源码、通信协议等，工程师们可根据不同需求进行修改开发，以适应更多的应用场景。