Future Technology Devices International Ltd.

Application Note AN_129

Interfacing FTDI USB Hi-Speed Devices to a JTAG TAP

Document Reference No.: FT000183

Version 1.1

Issue Date: 2011-09-02

This application note describes the use of the FTDI USB Hi-Speed FT232H, FT2232H and FT4232H devices to emulate a JTAG interface using their MPSSE

Future Technology Devices International Limited (FTDI)

Unit 1,2 Seaward Place, Glasgow G41 1HH, United Kingdom
Tel.: +44 (0) 141 429 2777 Fax: + 44 (0) 141 429 2758
E-Mail (Support): support1@ftdichip.com Web: http://www.ftdichip.com

Table of Contents

1 Introduction

The FT2232H and FT4232H are the FTDI's first USB 2.0 Hi-Speed (480Mbits/s) USB to UART/FIFO ICs. They also have the capability of being configured in a variety of serial interfaces using the internal MPSSE (Multi-Protocol Synchronous Serial Engine). The FT2232H device has two independent ports, both of which can be configured to use the MPSSE while only Channel A and B of FT4232H can be configured to use the MPSSE.

The FT232H, introduced in 2011, builds on the FTDI Hi-Speed USB family. The FT232H is a single-port UART/FIFO IC that has one MPSSE interface as well as several new modes.

Using the MPSSE can simplify the synchronous serial protocol (USB to SPI, I^2C, JTAG, etc.) design. This application note focuses on the hardware and software required to emulate a connection to a JTAG TAP test chain using the FT2232H. Users can use the example schematic and functional software code to begin their design. Note that software code listing is provided as an illustration only and not supported by FTDI. The FT232H and FT4232H can also be used with the example in this document, though pin-out and port selection will need to match the respective part.

The application example also duplicates the JTAG timing expected to be seen by the SN74BCT8244A to prove the function.

1.1 FTDI MPSSE Introduction

The Multi-Protocol Synchronous Serial Engine (MPSSE) is a feature of certain FTDI client ICs that allow emulation of several synchronous serial protocols including SPI, I2C and JTAG.

A single MPSSE is available in the FT2232D, a Full-Speed USB 2.0 client device. The FT2232D is capable of synchronous serial communication up to 6Mbps.

As noted above, two MPSSEs are available in the FT2232H and the FT4232H, both Hi-Speed USB 2.0 client devices. Each of the engines is capable of synchronous serial communications up to 30Mbps. The MPSSE in the FT2232H and FT4232H provide new commands for additional clock modes and is used in CPU interface and synchronous FIFO (parallel) modes. The FT232H contains a single MPSSE, the CPU Synchronous FIFO and the new FT1248 modes. Application note AN_135, MPSSE Basics and AN_167, FT1248 Dynamic Parallel/Serial Interface Basics provide more information on these other modes.

This application note describes the use of the MPSSE to emulate a JTAG interface. There are multiple references to AN_108 - Command Processor for MPSSE and MCU Host Bus Emulation Modes, also available from the FTDI Web Site.

1.2 JTAG background

Today's electronic circuits consist of numerous complex integrated circuits. A typical embedded system can contain multiple CPUs, programmable devices, memory, etc. With such complexity, it is often impossible to directly probe and test the entire functionality of a given design.

In 1990, the Institute of Electrical and Electronics Engineers (IEEE) ratified the standard 1149.1, which was the work of the Joint Test Action Group (JTAG). This standard defines a common means of implementing boundary-scan test functionality in an integrated circuit. It allows devices from different vendors to be present in a common chain to provide access to all of the Input and Output (I/O) pins. Commonly used with additional facilities, such as a bed-of-nails device, it is possible to perform functional and manufacturing tests on an entire circuit. It is common to refer to the IEEE 1149.1 standard as the "JTAG standard". Many published documents and articles use these terms interchangeably.

The IEEE 1149.1 was most recently updated in 2001. Additional IEEE standards reference 1149.1 while providing expanded features such as analog circuit tests in addition to digital circuit tests. These additional standards are 1149.4 - Analog Boundary Scan, 1149.6 - Advanced I/O and 1532 - In System Configuration. The latter is commonly used for programming memory devices and configuring programmable digital logic such as FPGAs and CPLDs.

JTAG (IEEE 1149.1) defines a synchronous state machine consisting of 16 states as noted in Figure 1.1.

JTAG TAP State Machine

Transitions on state of TMS on positive edge of TCK

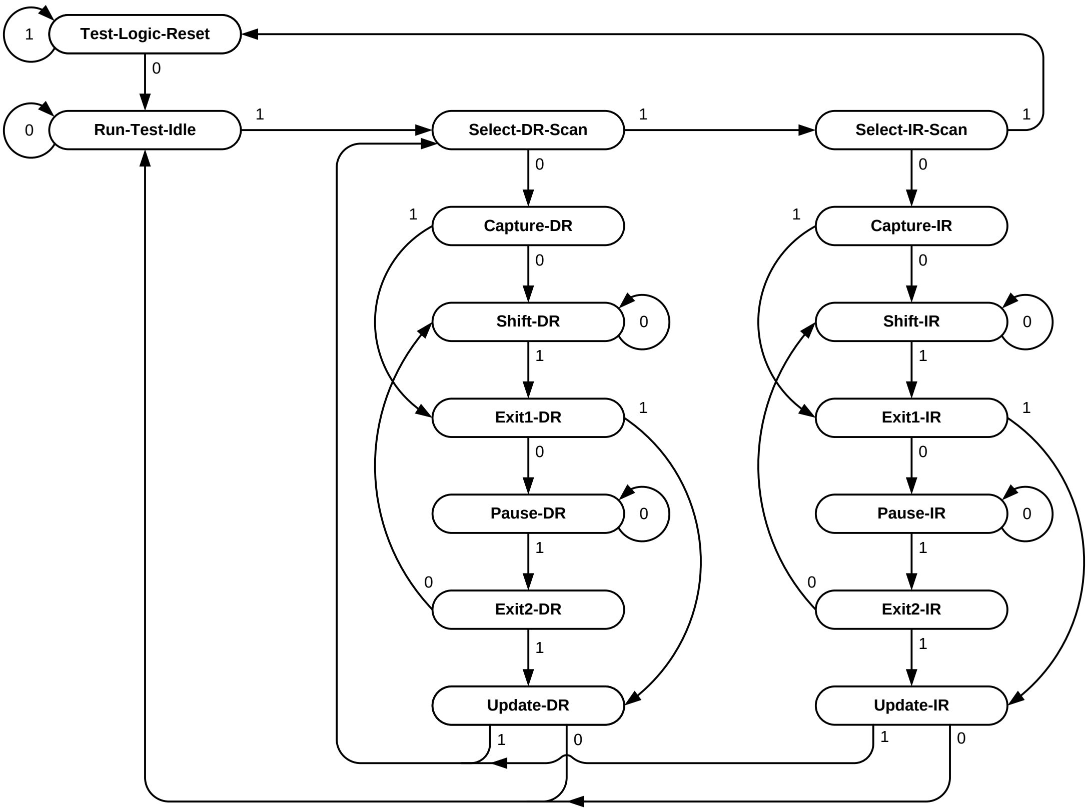

Figure 1.1 – IEEE 1149.1 (JTAG) state machine

The boundary scan circuitry is accessed through a Test Access Port (TAP) controller with four dedicated and mandatory I/O signals: Test Clock (TCK) - the input clock for the state machine, Test Mode Select (TMS) - the input used to navigate through the state machine, Test Data In (TDI) - the input containing serial data or instructions and Test Data Out (TDO) - the output containing serial data or instructions. An optional fifth signal, Test Reset (TRST#) can be implemented on a TAP. TRST# is an asynchronous reset that forces the state machine immediately to the Test-Logic-Reset state. It is important to note that even without TRST#, the state machine can always be forced to Test-Logic-Reset from any other state by holding TMS high for a maximum of five clock cycles.

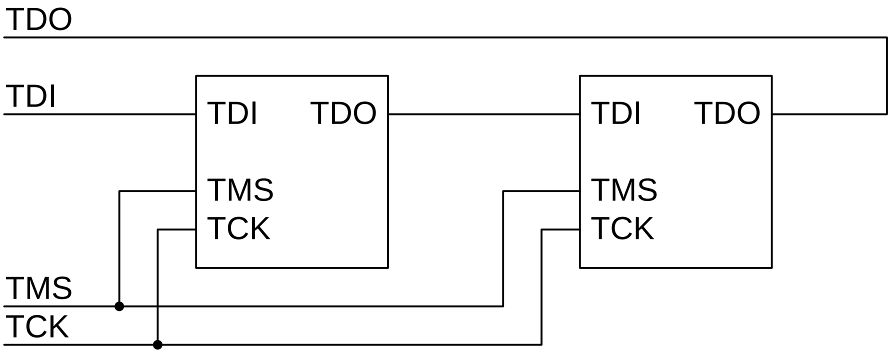

Figure 1.2 – IEEE 1149.1 (JTAG) TAP chain

As shown in Figure 1.2, devices in a JTAG chain share TCK and TMS. This forces all devices on a single chain to be in the same state within the state machine. The JTAG master controller connects its data output to TDI. Each device in the chain connects its TDI to the previous TDO. Finally, the last device in the chain connects its TDO to the controllers data input. Other connection schemes are possible; however, they are beyond the scope of this application note.

1.2.1 JTAG signalling

The IEEE 1149.1 specification identifies state transitions based on the state of TMS at the rising edge of TCK. Loading of instruction and data stimulus registers within the TAP as well as data shifting into TDI and out of TDO are also performed on the rising edge of TCK. The falling edge of TCK is used to latch data responses into the available registers in the boundary scan device. The registers within each JTAG TAP have different widths. It is important to maintain the level of TMS while data is shifted into and/or out of the registers.

The SN74BCT8244A contains the following JTAG TAP registers:

Register	Size
Instruction	8-bit
Boundary-Scan	18-bit
Boundary-Scan Control	2-bit
Bypass	1-bit

Table 1.1 – SN74BCT8244 JTAG TAP registers

If there are multiple devices in a TAP chain each register type can be of a different length for each of the devices. The JTAG master control program must account for these. There are six states throughout the JTAG state diagram that are designed to accommodate different devices with different register lengths. Referring to Figure 1.1, these states are: Test-Logic-Reset, Run-Test-Idle, Shift-DR, Pause-DR, Shift-IR and Pause-IR. Holding TMS at the appropriate value holds the state machine in the required state until valid bits are clocked to all registers for all devices in the TAP chain.

2 Example Circuit

A simple integrated circuit with a JTAG TAP is the Texas Instruments SN74BCT8244A (www.ti.com). This device consists of an octal buffer with two output enable pins and a JTAG TAP to provide the boundary scan capability. For this example, the FT2232H Mini Module will be used as shown in the circuit excerpt in Figure 2.1. USB and power connection details can be found in the FT2232H Datasheet, FT2232H Mini-Module Datasheet and DLP-USB1232H Datasheet.

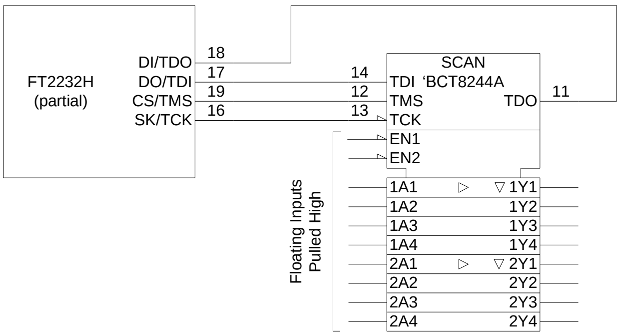

Figure 2.1 – Example circuit

When using the MPSSE, four pins of the FT2232H are defined for the synchronous serial interface. In addition to the FT2232H itself, two modules that utilize the FT2232H are also listed along with the corresponding pins.

JTAG Function	FT2232H IC Port A Pin Number	FT2232H Mini Module Pin Number	DLP-USB1232H Pin Number
TCK (output)	16	CN2-7	18
TDI (output)	17	CN2-10	16
TDO (input)	18	CN2-9	2
TMS (output)	19	CN2-12	5

Table 2.1 – FT2232H JTAG pin assignments

TDI and TDO appear to be reversed; however, these are the correct signal names as referenced by the JTAG TAP. The input pins of the SN74BCT8244A are internally pulled high. For this example circuit, they are left open. This fixes the input values at logic "1" and forces the outputs into a high-impedance state.

For this application note, Port A of the FT2232H is connected to the SN74BCT8244A. With the FT2232H and FT4232H, Port B could be used instead. In conjunction with the Port B pin assignments, the application program (see next section) would also require modification to access the MPSSE for port B.

TRST# is supported on the SN74BCT8244A; however, it requires an input of 10V on the TMS pin. To simplify the circuit, TRST# is not implemented in this example. Note that on a TAP with a standard I/O voltage, one of the unused GPIO pins of the FT2232H could be used for this function.

The FT2232H requires a VCCIO of 3.3V, although its inputs are 5V tolerant. This allows a direct connection with the 5V SN74BCT8244A. Inspection of the two datasheets will show the logic high and low input thresholds are indeed satisfied as well as maximum voltages not exceeded.

3 Example Program

The timing example on Page 14 of the Texas Instruments SN74BCT8244A datasheet is duplicated and the resultant data observed. This example consists of 25 cycles of TCK. All states of the JTAG TAP controller are utilized, with the exception of Pause-IR, Exit2-IR, Pause-DR and Exit2-DR. These unused states are typically only needed when a device has a longer JTAG chain, or very large Boundary-Scan registers.

This example program utilizes the FTDI D2XX device driver. It is written in a linear fashion to demonstrate the actual bytes being sent to the MPSSE and the resultant data read from the MPSSE. There are sections where reading and writing the data pins (TDI & TDO) must be combined with manipulating the control pin (TMS) in order to change states. The resultant data must be carefully observed and acted upon. Data may need shifted into a format that is more useful to the programmer.

In addition to duplicating the timing example, the Hi-Speed FTDI chips (FT2232H and FT4232H) support generation of TCK without clocking any data into our out of the MPSSE. This is demonstrated toward the end of the program listing. The code listing is followed by scope plots of the expected timing.

NOTE:

- The FT2232H and FT4232H require device driver version 2.06.00 or later.
- The FT232H requires device driver version 2.08.14 or later.
- In general, it is always a good idea to load the latest driver for all FTDI peripheral devices.

3.1 Code Listing

The example program is written in C++ and compiled in Microsoft® Visual Studio 2008 as a console application.

```
// AN_129_Hi-Speed_JTAG_with_MPSSE.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#include "ftd2xx.h"


int _tmain(int argc, _TCHAR* argv[])
{
    FT_HANDLE ftHandle;             // Handle of the FTDI device
    FT_STATUS ftStatus;             // Result of each D2XX call

    DWORD dwNumDevs;                // The number of devices
    unsigned int uiDevIndex = 0xF;  // The device in the list that is used

    BYTE byOutputBuffer[1024];      // Buffer to hold MPSSE commands and data to be sent to the FT2232H
    BYTE byInputBuffer[1024];       // Buffer to hold data read from the FT2232H

    DWORD dwCount = 0;              // General loop index
    DWORD dwNumBytesToSend = 0;     // Index to the output buffer
    DWORD dwNumBytesSent = 0;       // Count of actual bytes sent - used with FT_Write
    DWORD dwNumBytesToRead = 0;     // Number of bytes available to read in the driver's input buffer
    DWORD dwNumBytesRead = 0;       // Count of actual bytes read - used with FT_Read

    DWORD dwClockDivisor = 0x05DB;  // Value of clock divisor, SCL Frequency = 60/((1+0x05DB)*2) (MHz) = 20khz

    // Does an FTDI device exist?

    printf("Checking for FTDI devices...\n");

    ftStatus = FT_CreateDeviceInfoList(&dwNumDevs);
                                    // Get the number of FTDI devices
    if (ftStatus != FT_OK)          // Did the command execute OK?
    {
        printf("Error in getting the number of devices\n");
        return 1;                   // Exit with error
    }

    if (dwNumDevs < 1)              // Exit if we don't see any
    {
        printf("There are no FTDI devices installed\n");
        return 1;                   // Exist with error
    }

    printf("%d FTDI devices found - the count includes individual ports on a single chip\n", dwNumDevs);
```



```
// Open the port - For this application note, assume the first device is a FT2232H or FT4232H
// Further checks can be made against the device descriptions, locations, serial numbers, etc.
//      before opening the port.

printf("\nAssume first device has the MPSSE and open it...\n");

ftStatus = FT_Open(0, &ftHandle);
if (ftStatus != FT_OK)
{
    printf("Open Failed with error %d\n", ftStatus);
    return 1;                // Exit with error
}

// Configure port parameters

printf("\nConfiguring port for MPSSE use...\n");

ftStatus |= FT_ResetDevice(ftHandle);
                            //Reset USB device

//Purge USB receive buffer first by reading out all old data from FT2232H receive buffer
ftStatus |= FT_GetQueueStatus(ftHandle, &dwNumBytesToRead);
                            // Get the number of bytes in the FT2232H receive buffer
if ((ftStatus == FT_OK) && (dwNumBytesToRead > 0))
    FT_Read(ftHandle, &byInputBuffer, dwNumBytesToRead, &dwNumBytesRead);
                            //Read out the data from FT2232H receive buffer
ftStatus |= FT_SetUSBParameters(ftHandle, 65536, 65535);
                            //Set USB request transfer sizes to 64K
ftStatus |= FT_SetChars(ftHandle, false, 0, false, 0);
                            //Disable event and error characters
ftStatus |= FT_SetTimeouts(ftHandle, 0, 5000);
                            //Sets the read and write timeouts in milliseconds
ftStatus |= FT_SetLatencyTimer(ftHandle, 16);
                            //Set the latency timer (default is 16mS)
ftStatus |= FT_SetBitMode(ftHandle, 0x0, 0x00);
                            //Reset controller
ftStatus |= FT_SetBitMode(ftHandle, 0x0, 0x02);
                            //Enable MPSSE mode
if (ftStatus != FT_OK)
{
    printf("Error in initializing the MPSSE %d\n", ftStatus);
    FT_Close(ftHandle);
    return 1;                // Exit with error
}

Sleep(50); // Wait for all the USB stuff to complete and work

// -----------------------------------------------------------
// At this point, the MPSSE is ready for commands
// -----------------------------------------------------------
// -----------------------------------------------------------
// Synchronize the MPSSE by sending a bogus opcode (0xAA),
//      The MPSSE will respond with "Bad Command" (0xFA) followed by
//      the bogus opcode itself.
// -----------------------------------------------------------

byOutputBuffer[dwNumBytesToSend++] = 0xAA;//'\xAA';
                            //Add bogus command 'xAA' to the queue
ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                            // Send off the BAD commands
dwNumBytesToSend = 0;        // Reset output buffer pointer
do
{
    ftStatus = FT_GetQueueStatus(ftHandle, &dwNumBytesToRead);
                            // Get the number of bytes in the device input buffer
} while ((dwNumBytesToRead == 0) && (ftStatus == FT_OK));
                            //or Timeout

bool bCommandEchod = false;

ftStatus = FT_Read(ftHandle, &byInputBuffer, dwNumBytesToRead, &dwNumBytesRead);
                            //Read out the data from input buffer
for (dwCount = 0; dwCount < dwNumBytesRead - 1; dwCount++)
                            //Check if Bad command and echo command received
{
    if ((byInputBuffer[dwCount] == 0xFA) && (byInputBuffer[dwCount+1] == 0xAA))
    {
        bCommandEchod = true;
        break;
    }
}
if (bCommandEchod == false)
{
    printf("Error in synchronizing the MPSSE\n");
    FT_Close(ftHandle);
    return 1;                // Exit with error
}

// -----------------------------------------------------------
// Configure the MPSSE settings for JTAG
//      Multple commands can be sent to the MPSSE with one FT_Write
// -----------------------------------------------------------

dwNumBytesToSend = 0;        // Start with a fresh index

// Set up the Hi-Speed specific commands for the FTx232H

byOutputBuffer[dwNumBytesToSend++] = 0x8A;
                            // Use 60MHz master clock (disable divide by 5)
byOutputBuffer[dwNumBytesToSend++] = 0x97;
                            // Turn off adaptive clocking (may be needed for ARM)
byOutputBuffer[dwNumBytesToSend++] = 0x8D;
                            // Disable three-phase clocking
ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                            // Send off the HS-specific commands
```

```
dwNumBytesToSend = 0;        // Reset output buffer pointer

// Set initial states of the MPSSE interface - low byte, both pin directions and output values
//      Pin name        Signal  Direction       Config  Initial State       Config
//      ADBUS0      TCK     output      1       low             0
//      ADBUS1      TDI     output      1       low             0
//      ADBUS2      TDO     input       0                       0
//      ADBUS3      TMS     output      1       high                1
//      ADBUS4      GPIOL0  input       0                       0
//      ADBUS5      GPIOL1  input       0                       0
//      ADBUS6      GPIOL2  input       0                       0
//      ADBUS7      GPIOL3  input       0                       0

byOutputBuffer[dwNumBytesToSend++] = 0x80;
                            // Set data bits low-byte of MPSSE port
byOutputBuffer[dwNumBytesToSend++] = 0x08;
                            // Initial state config above
byOutputBuffer[dwNumBytesToSend++] = 0x0B;
                            // Direction config above

ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                            // Send off the low GPIO config commands

dwNumBytesToSend = 0;        // Reset output buffer pointer

// Set initial states of the MPSSE interface - high byte, both pin directions and output values
//      Pin name        Signal  Direction       Config  Initial State       Config
//      ACBUS0      GPIOH0  input       0                       0
//      ACBUS1      GPIOH1  input       0                       0
//      ACBUS2      GPIOH2  input       0                       0
//      ACBUS3      GPIOH3  input       0                       0
//      ACBUS4      GPIOH4  input       0                       0
//      ACBUS5      GPIOH5  input       0                       0
//      ACBUS6      GPIOH6  input       0                       0
//      ACBUS7      GPIOH7  input       0                       0

byOutputBuffer[dwNumBytesToSend++] = 0x82;
                            // Set data bits low-byte of MPSSE port
byOutputBuffer[dwNumBytesToSend++] = 0x00;
                            // Initial state config above
byOutputBuffer[dwNumBytesToSend++] = 0x00;
                            // Direction config above

ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                            // Send off the high GPIO config commands

dwNumBytesToSend = 0;        // Reset output buffer pointer

// Set TCK frequency
// TCK = 60MHz /((1 + [(1 +0xValueH*256) OR 0xValueL])*2)

byOutputBuffer[dwNumBytesToSend++] = '\x86';
                            //Command to set clock divisor
byOutputBuffer[dwNumBytesToSend++] = dwClockDivisor & 0xFF;
                            //Set 0xValueL of clock divisor
byOutputBuffer[dwNumBytesToSend++] = (dwClockDivisor >> 8) & 0xFF;
                            //Set 0xValueH of clock divisor
ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                            // Send off the clock divisor commands

dwNumBytesToSend = 0;        // Reset output buffer pointer


// Disable internal loop-back

byOutputBuffer[dwNumBytesToSend++] = 0x85;
                            // Disable loopback

ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                            // Send off the loopback command

dwNumBytesToSend = 0;        // Reset output buffer pointer



// Navigage TMS through Test-Logic-Reset -> Run-Test-Idle -> Select-DR-Scan -> Select-IR-Scan
//                      TMS=1                TMS=0            TMS=1              TMS=1

byOutputBuffer[dwNumBytesToSend++] = 0x4B;
                            // Don't read data in Test-Logic-Reset, Run-Test-Idle, Select-DR-Scan, Select-IR-Scan
byOutputBuffer[dwNumBytesToSend++] = 0x05;
                            // Number of clock pulses = Length + 1 (6 clocks here)
byOutputBuffer[dwNumBytesToSend++] = 0x0D;
                            // Data is shifted LSB first, so the TMS pattern is 101100
ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                            // Send off the TMS command
dwNumBytesToSend = 0;        // Reset output buffer pointer

// TMS is currently low.
// State machine is in Shift-IR, so now use the TDI/TDO command to shift 1's out TDI/DO while reading TDO/DI
// Although 8 bits need shifted in, only 7 are clocked here.  The 8th will be in conjunciton with a TMS command, coming next

byOutputBuffer[dwNumBytesToSend++] = 0x3B;
                            // Clock data out throuth states Capture-IR, Shift-IR and Exit-IR, read back result
byOutputBuffer[dwNumBytesToSend++] = 0x06;
                            // Number of clock pulses = Length + 1 (7 clocks here)
byOutputBuffer[dwNumBytesToSend++] = 0xFF;
                            // Shift out 1111111 (ignore last bit)
ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                            // Send off the TMS command
dwNumBytesToSend = 0;        // Reset output buffer pointer
```

```
// Here is the TMS command for one clock.  Data is also shifted in.

byOutputBuffer[dwNumBytesToSend++] = 0x6B;
                            // Clock out TMS, Read one bit.
byOutputBuffer[dwNumBytesToSend++] = 0x00;
                            // Number of clock pulses = Length + 0 (1 clock here)
byOutputBuffer[dwNumBytesToSend++] = 0x83;
                            // Data is shifted LSB first, so TMS becomes 1.  Also, bit 7 is shifted into TDI/DO, also a 1
                            // The 1 in bit 1 will leave TMS high for the next commands.
ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                            // Send off the TMS command
dwNumBytesToSend = 0;       // Reset output buffer pointer


// Navigage TMS from Exit-IR through Update-IR -> Select-DR-Scan -> Capture-DR
//                                   TMS=1        TMS=1             TMS=0

byOutputBuffer[dwNumBytesToSend++] = 0x4B;
                            // Don't read data in Update-IR -> Select-DR-Scan -> Capture-DR
byOutputBuffer[dwNumBytesToSend++] = 0x03;
                            // Number of clock pulses = Length + 1 (4 clocks here)
byOutputBuffer[dwNumBytesToSend++] = 0x83;
                            // Data is shifted LSB first, so the TMS pattern is 110
ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                            // Send off the TMS command
dwNumBytesToSend = 0;       // Reset output buffer pointer


// TMS is currently low.
// State machine is in Shift-DR, so now use the TDI/TDO command to shift 101 out TDI/DO while reading TDO/DI
// Although 3 bits need shifted in, only 2 are clocked here.  The 3rd will be in conjunciton with a TMS command, coming next

byOutputBuffer[dwNumBytesToSend++] = 0x3B;
                            // Clock data out throuth states Shift-DR and Exit-DR.
byOutputBuffer[dwNumBytesToSend++] = 0x01;
                            // Number of clock pulses = Length + 1 (2 clocks here)
byOutputBuffer[dwNumBytesToSend++] = 0x01;
                            // Shift out 101 (ignore last bit)
ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                            // Send off the TMS command
dwNumBytesToSend = 0;       // Reset output buffer pointer


// Here is the TMS command for one clock.  Data is also shifted in.

byOutputBuffer[dwNumBytesToSend++] = 0x6B;
                            // Clock out TMS, Read one bit.
byOutputBuffer[dwNumBytesToSend++] = 0x00;
                            // Number of clock pulses = Length + 0 (1 clock here)
byOutputBuffer[dwNumBytesToSend++] = 0x83;
                            // Data is shifted LSB first, so TMS becomes 1.  Also, bit 7 is shifted into TDI/DO, also a 1
                            // The 1 in bit 1 will leave TMS high for the next commands.
ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                            // Send off the TMS command
dwNumBytesToSend = 0;       // Reset output buffer pointer


// Navigage TMS through Update-DR -> Select-DR-Scan -> Select-IR-Scan -> Test Logic Reset
//                      TMS=1         TMS=1             TMS=1             TMS=1
byOutputBuffer[dwNumBytesToSend++] = 0x4B;
                            // Don't read data in Update-DR -> Select-DR-Scan -> Select-IR-Scan -> Test Logic Reset
byOutputBuffer[dwNumBytesToSend++] = 0x03;
                            // Number of clock pulses = Length + 1 (4 clocks here)
byOutputBuffer[dwNumBytesToSend++] = 0xFF;
                            // Data is shifted LSB first, so the TMS pattern is 101100
ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                            // Send off the TMS command
dwNumBytesToSend = 0;       // Reset output buffer pointer

do
{
    ftStatus = FT_GetQueueStatus(ftHandle, &dwNumBytesToRead);
                            // Get the number of bytes in the device input buffer
} while ((dwNumBytesToRead == 0) && (ftStatus == FT_OK));
                            //or Timeout

ftStatus = FT_Read(ftHandle, &byInputBuffer, dwNumBytesToRead, &dwNumBytesRead);
                            //Read out the data from input buffer


printf("\n");
printf("TI SN74BCT8244A IR default value is 0x81\n");
printf("The value scanned by the FT2232H is 0x%x\n", byInputBuffer[dwNumBytesRead - 3]);
printf("\n");
printf("TI SN74BCT8244A DR bypass expected data is 00000010 = 0x2\n");
printf("                   The value scanned by the FT2232H = 0x%x\n", (byInputBuffer[dwNumBytesRead-1] >> 5));

// Generate a clock while in Test-Logic-Reset
// This will not do anything with the TAP in the Test-Logic-Reset state,
//      but will demonstrate generation of clocks without any data transfer

byOutputBuffer[dwNumBytesToSend++] = 0x8F;
                            // Generate clock pulses
byOutputBuffer[dwNumBytesToSend++] = 0x02;
                            // (0x0002 + 1) * 8 = 24 clocks
byOutputBuffer[dwNumBytesToSend++] = 0x00;
                            //

ftStatus = FT_Write(ftHandle, byOutputBuffer, dwNumBytesToSend, &dwNumBytesSent);
                            // Send off the clock commands

dwNumBytesToSend = 0;       // Reset output buffer pointer
```

```
    /*
    // -----------------------------------------------------------
    // Start closing everything down
    // -----------------------------------------------------------
    */

    printf("\nJTAG program executed successfully.\n");
    printf("Press <Enter> to continue\n");
    getchar();                  // wait for a carriage return

    FT_Close(ftHandle);         // Close the port

    return 0;                   // Exit with success
}
```

3.2 Program Output

The Texas Instruments example timing diagram is duplicated with an oscilloscope screen image:

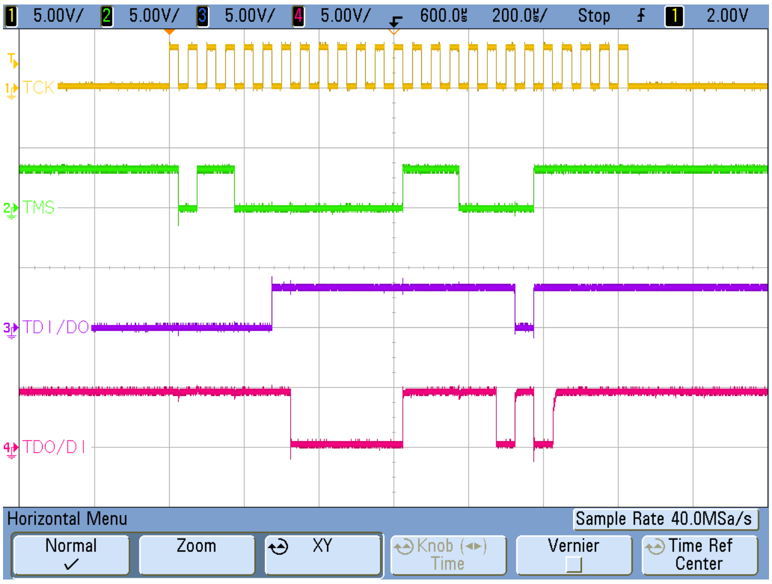

Figure 3.1 – SN74BCT8244A timing example observation

Note that TDI/DO is always driven, and TDO/DI is pulled high by the SN74BCT8244A. The Texas Instruments datasheet indicates several areas of "don't care" which end up as logic "1" in this screen shot.

TCK is generated without any activity on TDI, TDO or TMS.

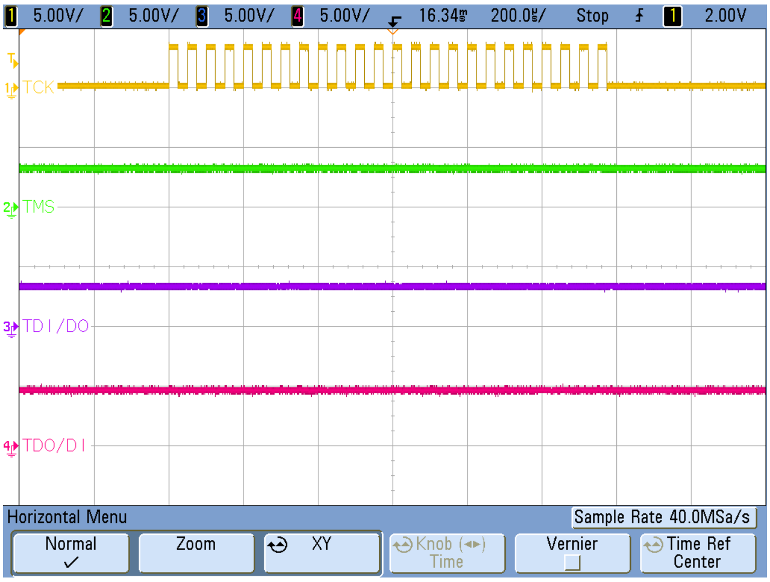

Figure 3.2 – TCK generation

This is useful to run an internal test within a particular TAP. There are several options available which include a specific number of pulses, or to pulse until a GPIO signal is set to a known value. In this example, 24 clocks are generated.

4 Summary

The circuit and application program described in this application note demonstrate the basics of establishing communication with the MPSSE, configured for JTAG, on the FT2232H. In particular, the timing diagram shown in the SN74BCT8244A datasheet is duplicated through an example program utilizing the FTDI D2XX device driver to prove a known result.

As mentioned, the information in this application note also covers the FT232H and FT4232H with appropriate modifications to account for port selection and pin assignments.

5 Acronyms and Abbreviations

Terms	Description
MPSSE	Multi-Protocol Synchronous Serial Engine – a state machine in certain FTDI USB client devices that can be used to emulate serial protocols such as SPI, I2C and JTAG
JTAG	Joint Test Action Group – An industry organization responsible for generating a standard for in-circuit testing of complex circuits. JTAG is also commonly used in place of the full specification name IEEE 1149.1
IEEE	Institute of Electrical and Electronics Engineers
IEEE 1149.1	Commonly referred to as "JTAG" – Industry standard describing building blocks and software used to provide in-circuit test capabilities

Table 5.1 – Acronyms and Abbreviations

6 Contact Information

Head Office – Glasgow, UK

Future Technology Devices International Limited
Unit 1, 2 Seaward Place, Centurion Business Park
Glasgow G41 1HH
United Kingdom
Tel: +44 (0) 141 429 2777
Fax: +44 (0) 141 429 2758

E-mail (Sales)	sales1@ftdichip.com
E-mail (Support)	support1@ftdichip.com
E-mail (General Enquiries)	admin1@ftdichip.com
Web Site URL	http://www.ftdichip.com
Web Shop URL	http://www.ftdichip.com

Branch Office – Hillsboro, Oregon, USA

Future Technology Devices International Limited (USA)
7235 NW Evergreen Parkway, Suite 600
Hillsboro, OR 97123-5803
USA
Tel: +1 (503) 547 0988
Fax: +1 (503) 547 0987

E-Mail (Sales)	us.sales@ftdichip.com
E-Mail (Support)	us.support@ftdichip.com
E-Mail (General Enquiries)	us.admin@ftdichip.com
Web Site URL	http://www.ftdichip.com

Branch Office – Taipei, Taiwan

Future Technology Devices International Limited (Taiwan)
2F, No. 516, Sec. 1, NeiHu Road
Taipei 114
Taiwan , R.O.C.
Tel: +886 (0) 2 8791 3570
Fax: +886 (0) 2 8791 3576

E-mail (Sales)	tw.sales1@ftdichip.com
E-mail (Support)	tw.support1@ftdichip.com
E-mail (General Enquiries)	tw.admin1@ftdichip.com
Web Site URL	http://www.ftdichip.com

Branch Office – Shanghai, China

Future Technology Devices International Limited (China)
Room 408, 317 Xianxia Road,
Shanghai, 200051
China
Tel: +86 21 62351596
Fax: +86 21 62351595

E-mail (Sales)	cn.sales@ftdichip.com
E-mail (Support)	cn.support@ftdichip.com
E-mail (General Enquiries)	cn.admin@ftdichip.com
Web Site URL	http://www.ftdichip.com

Distributor and Sales Representatives

Please visit the Sales Network page of the FTDI Web site for the contact details of our distributor(s) and sales representative(s) in your country.

Appendix A - References

- FT2232H Datasheet FTDI Ltd.
- AN_108 Command Processor For MPSSE and MCU Host Bus Emulation Modes, Version 1.2 FTDI Ltd.
- AN_135 MPSSE Basics, FTDI Ltd.
- AN_167 FT1248 Dynamic Parallel/Serial Interface Basics, FTDI Ltd.
- Texas Instruments SN72BCT8244A Datasheet, © 1990, 1996
- The Boundary-Scan Handbook, 3rd Ed., Kenneth P. Parker, © 2003, Kluwer Academic Publishers, ISBN 1-4020-7496-4

Appendix B - List of Figures and Tables

List of Figures

List of Tables

Appendix C - Revision History

Revision History

Version 1.0	Initial Release	20th October, 2009
Version 1.1	Added references to FT232H	2nd September, 2011