Tools Tech Note 3
SdJtag Emulation Interface

Document Revision 0.05
Oct 14, 2004

Introduction

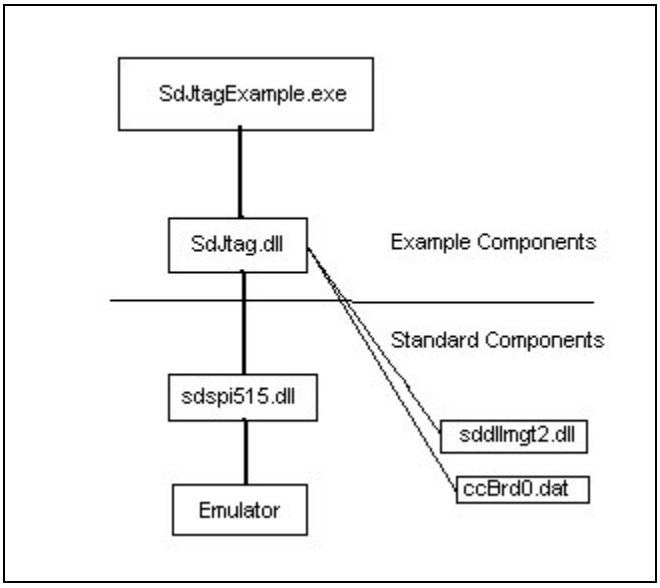
This tech note describes a simple "C" based interface to Spectrum Digital (SD) emulators which is called ***SdJtag***. The purpose of this interface is to enable knowledgeable programmers access to basic functions of SD emulators. The interface is a thin wrapper around the standard SD emulation drivers shipped as a component to TI's CCS product. All emulation drivers are available from the SD ftp site.

SdJtag Example Package

The SdJtag example package includes the necessary components to build and link to SD emulation drivers installed as part of a CCS installation. This package should be installed into your default CCS installation directory and "SpecDig" subdirectory, typically "c:\ti\specdig\sdjtag". The following items will be installed:

- ExampleSdJtag.dsp, ExampleSdJtag.dsw: MSVC 6.0 Project files.
- ExampleSdJtag.exe: Binary for the example.
- ExampleSdJtag.c: The example code.
- Sdjtag.h: Interface header file to SdJtag.dll.
- Sdjtag.lib: Export lib for SdJtag.dll.
- SdJtag.dll: The SdJtag wrapper for SD emulators.
- Usbjtag.dll: XDS510USB specific wrapper.
- CcBrdARM7.dat, ccBrdARM9.dat, and dummy.dat: CCS example board.dat file for testing.
- TechNote_3.pdf. This document.

The figure below shows the basic example components and standard components in an emulator/CCS install.



The example will highlight the following operations:

- Opening an emulator connection.
- Checking emulator status.
- Recovering from target power loss/cycle.
- Calling JTAG scan functions to find the JTAG instruction register length and number of devices in the scan chain.
- Closing an emulator connection.

The header file sdjtag.h includes specific details for all data types and specifics on all functions. The header file should be used as the main source of documentation for sdjtag. The examples below serve only a means to tie all the pieces together.

Connection Rules

There are some basic connection rules that must be followed with using the simply sdjtag interface:

1. Only one process may make a connection to sdjtag.dll and the underlying emulation dlls.
2. No other tool such as CCS, SDFlash, Flash470 etc. may be running while there is a connection to sdjtag.dll.
3. Do NOT assume you know how to correlate board file cpu names with device ids. Let the interface provide the proper id based on the cpu name.
4. When in a multiprocessor environment it is best to target one cpu, do a unit of work then target the next cpu if required. This mimics the way CCS and other apps operate and reduce the likely hood of treading on untested emulation sequence interaction. It also reduces the amount of thrashing that must go on in low level emulation drivers to switch between devices.
5. Do NOT abort a connection. Instead close the connection and free handles if they are valid. The helps to prevent possible memory leaks, open handles, etc..

Opening the Emulation Connection

To open an emulation connection there are several pieces of information that must be obtained from SdConfig and CC setup. From SdConfig you need to know, 1) Emulator Port Address, 2) Name of hardware product. Both items are available from the "Emu" panel of the selected emulator. From CC setup you need to know, 1) path to the CC setup generated board file, 2) name of your target cpu in the board file. By default CCS will name the board file ***ccBrd0.dat*** and place this in the <CCS_INSTALL_DIR>\cc\bin\BrdDat directory. The following example shows the contents of a single processor ARM7 board file. The target cpu name is "cpu_0".

```
;CFG-2.0
"cpu_0"   TIARM7xx
```

The following example shows a board file for two ARM7 devices and two bypass devices.

```
;CFG-2.0
"cpu_0"   BYPASS008
"cpu_1"   TIARM7xx
"cpu_2"   BYPASS008
"cpu_3"   TIARM7xx
```

Two things to note in a multiprocessor setup:

1. All processor are named "cpu_x" no matter how you name them in CC setup.
2. The order of processors in the board file is inverted from the view in CC setup. The TI board file format specifies that the device nearest the emulator TDO pin is the first device in the scan chain which would be "cpu_0". The last device in the scan chain is near emulator TDI which corresponds to "cpu_3". For example:
   EMULATOR TDI->CPU_3->CPU_2->CPU_1->CPU_0->EMULATOR TDO->

With this information in hand opening the connection to the emulator is a two-step process:

1. Get a handle for the emulation session. In this step sdjtag.dll will locate all the needed emulation drivers components and load any underlying dlls as required. However no contact will be made to the emulator or target until the next step.
2. Make the physical connection to emulator and target. In this step the physical connection to the emulator and target is made. The emulator will be configured, checks will be made for good target power, TRSTn will be pulsed or taken high to enter the emulation mode. If successful a device id will be returned. This device id corresponds to the position of the target device in the scan chain and is derived from the cpu name, i.e. "cpu_0". Following a successful return the target device should be in a emulation ready state and can accept JTAG operations.

```
// Set the default arguments
SDJTAG_HNDL     Hndl       = NULL;
SDJTAG_EMU_TYPE EmuType    = SDJTAG_TYPE_XDS510PP_PLUS;
SDJTAG_PORT_ID  EmuId      = 0x378;
SDJTAG_DEV_ID   DevId      = -1;

int             RetryCount = 0;

// CCS default board file path
char ccBrd0Path[MAX_PATH+2] = "c:\\ti\\cc\\bin\\brddat\\ccBrd0.dat";

// CCS default device name for single processor scan chain
char ccDevName[34]  = "cpu_0";


 // Create a handle for our session
 Error = SDJTAG_CreateHndl( &Hndl, EmuType, EmuId, NULL, ccBrd0Path );
 if( ( Error != SDJTAG_ERR_NONE ) || ( Hndl == NULL ))
       goto EXIT_main;

 // After this point we must release handle before the exit
 Error = SDJTAG_Connect(  Hndl, ccDevName, &DevId );
 if( ( Error != SDJTAG_ERR_NONE ) || ( DevId == -1 ))
    goto EXIT_main;
```

Checking Emulator Status and Recovering from Target Power Loss/Cycle

Following the emulation connection a status check should be made to double check target conditions. Optionally a status check should be made following any command execution errors. The following code segment makes a status check following connection:

```
   // Following an emulator connect we should be in a good state to
   // perform emulation.  By definition SD emulators should clear
   // target power conditions, and command busy conditions.  A missing
   // TCK or TRSTn low is not an error condition. The user could be
   // controlling TCK externally.  TRSTn would normally be high but
   // the emulators do not necessarily read the TRSTn pin. On some
   // emulators both TCK and TRSTn have configuration controls in
   // sdopts.cfg.  See the sdopts.cfg file for details.
   //
   do
   {
      Error = SDJTAG_Status( Hndl, DevId, &Status );
      if(    ( Error != SDJTAG_ERR_NONE )
         || ( Status & SDJTAG_STAT_SAW_TPOWER_CYCLE )
         || ( Status & SDJTAG_STAT_NO_TPOWER )
         || ( Status & SDJTAG_STAT_CMD_BUSY ))
      {
         // You could post a user message and ask that they take some
         // action.  If action is to retry then first disconnect then
         // attempt to reconnect.
         //
         if( RetryCount != 0 )
         {
            SDJTAG_Disconnect( Hndl, DevId );
            // Sleep for 1 second to create delay
            Sleep( 1000 );
            DevId = -1;
```

```
            Error = SDJTAG_Connect(  Hndl, ccDevName, &DevId );
            if( ( Error != SDJTAG_ERR_NONE ) || ( DevId == -1 ))
               goto EXIT_main;

            RetryCount--;
         }
         else
            goto EXIT_main;
      }
   }
   while( RetryCount > 0 );
```

In this example we assume that the emulation connection may fail on startup as there is no target power, or the JTAG cable is disconnected etc.. When this occurs disconnect from the emulator, sleep or inform the user then reconnect. We will continue trying to make the connection until the retry counter expires.

JTAG Scan Functions

The JTAG functions are more completely documented in the sdjtag.h file but the following is a simple example of how you might find the full IR length of a scan chain.

```
int
FindIrLength( int Pattern )
{
   int IrLength = 0;
   int i,j;
   unsigned short ReadVal;
   unsigned short FillPattern;
   unsigned short MaskPattern;
   unsigned short BitVal = Pattern & 1;
   SDJTAG_SCAN    XmitArgs;
   SDJTAG_SCAN    RecvArgs;


   if( Pattern & 1 )
   {
      FillPattern = 0x0000;
      MaskPattern = 0x8000;
   }
   else
   {
      FillPattern = 0xFFFF;
      MaskPattern = 0x7FFF;
   }

   memset( XmitBuf,FillPattern, sizeof(XmitBuf));

   memset( RecvBuf,0, sizeof(RecvBuf));

   XmitBuf[SCAN_BUFFER_SIZE-1] = MaskPattern;

   XmitArgs.NumIntrBits = (SCAN_BUFFER_SIZE*NUMBITS_IN_VCT_BUF);
   XmitArgs.pXmit       = XmitBuf;
   XmitArgs.pRecv       = RecvBuf;
   XmitArgs.EndState    = IPAUS;
   SDJTAG_ScanIr( Hndl, DevId, &XmitArgs );

   RecvArgs.NumIntrBits = (SCAN_BUFFER_SIZE*NUMBITS_IN_VCT_BUF);
   RecvArgs.pXmit       = XmitBuf;
   RecvArgs.pRecv       = RecvBuf;
   RecvArgs.EndState    = IPAUS;
   SDJTAG_ScanIr( Hndl, DevId, &RecvArgs );
```

```
   // Always put the JTAG IR back in IR-BYPASS
   memset( XmitBuf,0xFFFF, sizeof(XmitBuf));
   XmitArgs.NumIntrBits = (SCAN_BUFFER_SIZE*NUMBITS_IN_VCT_BUF);
   XmitArgs.pXmit       = XmitBuf;
   XmitArgs.pRecv       = RecvBuf;
   XmitArgs.EndState    = IPAUS;
   SDJTAG_ScanIr( Hndl, DevId, &XmitArgs );

   for( i=0; i<SCAN_BUFFER_SIZE; i++ )
   {
      ReadVal = RecvBuf[i];
      for( j=0; j<16; j++ )
      {
         IrLength++;
         // Look for the first Pattern in the scan chain
         if( (ReadVal & 1 ) == BitVal )
            return( IrLength );

         ReadVal = ReadVal >> 1;
      }
   }
   return( 0 );
}
```

Several basic rules for all JTAG commands:

1. Set all elements in the structure. If not used then set to 0 or NULL.
2. Always check the return value for success before executing the next function.
3. Treat elements of the structure as volatile. Although lower levels of emulation drivers do not normally modify input parameters there is no rule that prevents this.
4. If there is anderror you can make a status check to see if this is due to power or command busy.
5. If the error returned is SDJTAG_ERR_CONTROLLER then it is recommended that you call SDJTAG_GetLastEmulatorError(). This will return the last “raw” error from the emulation controller, which can then be used by emulator vendor for more precise debug. Sdjtag does NOT attempt to map all emulator controller errors to sdjtag errors.
6. Once you have an “Error” number you can use the function SDJTAG_GetErrorMessage(Error) to obtain a error message string.

Closing the Connection

Closing the emulation connection is the reverse of opening the connection. If you have a good handle and good device id then you can disconnect from the target. If you have a good handle then you can free the handle. The sdjtag interface will check for NULL handles and valid device ids to prevent access to uninitialized interfaces. Sdjtag will attempt to close any open connections if the handle is freed prior to closing individual device connections.

```
   // Common exit point to do all the cleanup.  If we have a valid handle
   // then we can do the cleanup.
   if( Hndl != NULL )
   {
      // Only disconnect from a valid DevId
      if( DevId != -1 )
         SDJTAG_Disconnect( Hndl, DevId );

      // Not required but safe if doing multiple connect/disconnect
      DevId = -1;


      SDJTAG_FreeHndl( Hndl );
      // Not required but safe if doing multiple create/free handle
      Hndl = NULL;
   }
```

Scan Path Test versus Emulation Test

The sdjtag interface supports both scan path testing which is similar to SdConfig and emulation testing which is similar to CCS. However due to differences between the tools and history they have to be treated as two different operations. Normally scan path testing is done first to verify scan chain length and integrity then emulation testing is performed. Following is the suggested sequence:

1. Open a JTAG connection with only one device enumerated in the board data file. For example
   ```
   ;CFG-2.0
   "dummy"   TIARM7xx
   ```
   This fulfills the low-level emulation driver need to have at least one device in the board file. Also, by having only one device in the board file the low-level emulation driver will not perform background scan operations to "bypass" the unaddressed devices.
2. Perform scan path testing.
3. Close the connection and free the handle. It is necessary to free the handle to release the board file data so that a new board file can be selected.
4. Reopen the JTAG connection with a board file the fully describes the scan chain of your target. Now you can perform emulation testing of 1-N selected targets on the scan chain. The low-level emulation driver will now "bypass" the unaddressed devices. See **Limitations** section for XDS510USB specific limitations

XDS510USB Limitations and Special Requirements

The XDS510USB emulator does not expose a full low level emulation driver interface to it has limitations that do not apply to other emulators.

1. The sdjtag package MUST be installed in the <ccs_install_dir>\specdig subdirectory. The unzip will create the sdjtag subdirectory. XDS510USB support requires several components from the standard emulation driver install, located at <ccs_install_dir>\drivers. Sdjtag will look back two directory levels to find the "**drivers"** directory and the required files.

2. The XDS510USB emulation drivers do NOT expose a complete interface. Instead it only exposes a subset of functions to satisfy the needs of the SdConfig utility. The result is that a board file can only contain a single processor, i.e. the example above that uses the dummy.dat file. With this limitation you can only perform basic scan path testing. This would allow you to find the number of devices in the scan chain but does not allow you to specifically target a given device while ignoring the others. To get around this limitation the user has to take care of bypassing the other devices in the scan chain. This manual process of handling the bypassed devices would work across all emulators. An example of this is provided for reading the ARM silicon version. In version 6 of the file SdJtag.c a more complete example is provided for handling multiprocessors for ARM7/9 debug communications channel accesses.