The mpssespi Package

A Tcl Interface to the FTDI MSPPE SPI Library

Copyright © 2014 G. Andrew Mangogna

Legal Notice and Other Information

This software is copyrighted 2014 by G. Andrew Mangogna. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files.

The author hereby grants permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. Modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

GOVERNMENT USE: If you are acquiring this software on behalf of the U.S. government, the Government shall have only "Restricted Rights" in the software and related documentation as defined in the Federal Acquisition Regulations (FARs) in Clause 52.227.19 (c) (2). If you are acquiring the software on behalf of the Department of Defense, the software shall be classified as "Commercial Computer Software" and the Government shall have only "Restricted Rights" as defined in Clause 252.227-7013 (c) (1) of DFARs. Notwithstanding the foregoing, the authors grant the U.S. Government and others acting in its behalf permission to use and distribute the software in accordance with the terms specified in this license.

NOTICE OF FUTURE TECHNOLOGY DEVICES INTERNATIONAL LIMITED SOFTWARE

This software is linked against libraries provided by Future Technology Devices International Limited. The following is from the http://www.ftdichip.com/Drivers/D2XX.htm page of their web site.

This software is provided by Future Technology Devices International Limited ``as is” and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall future technology devices international limited be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

FTDI drivers may be used only in conjunction with products based on FTDI parts. FTDI drivers may be distributed in any form as long as license information is not modified. If a custom vendor ID and/or product ID or description string are used, it is the responsibility of the product manufacturer to maintain any changes and subsequent WHQL re-certification as a result of making these changes.

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
1.0b1	April 20, 2014	Initial release.	GAM
1.0	May 22, 2014	Official release of version 1.0.	GAM

Contents

Introduction 1

Design Concepts 1

Package Interface Considerations 1

Package Data 2

Package Commands 6

Get Number of Channels 6

Get Channel Info 7

Open Channel 11

Init Channel 12

Get Channel Configuration 14

Set Channel Configuration 20

Close Channel 28

Read Channel 30

Write Channel 34

Read/Write Channel 38

Is Busy 40

Change CS 42

Write GPIO 43

Read GPIO 45

Error Handling 47

Package Initialization 48

Load Initialization 48

Creating the Package Namespace 49

Unloading 49

Safe Interpreter Initialization 49

Package Configuration 50

Source Organization 50

Package Source 50

Test Source 52

Example 54

Special Linux Considerations 57

Known Problems 58

Building the Package 58

Index 59

List of Figures

1 Interface Layers 1

Abstract

This document is a literate program for the `mpssespi` Tcl package. As a literate program it contains both a discussion of the details of the package as well as the implementation source code. The `mpssespi` package provides a Tcl interface to the FTDI libMPSSE-SPI library. This library is used to interface FTDI USB to serial converter chips to SPI bus peripheral components allowing the peripherals to be controlled across a USB interface.

Introduction

Future Technology Devices International, Ltd. ® (FTDI) produces a series of USB to serial converter chips that are popular for interfacing a variety of systems across a USB bus. FDTI provides the necessary software drivers to interface to their chips. One of the features of FTDI converter chips is a Multi-Protocol Synchronous Serial Engine (MPSSE) that can be used to drive a variety of synchronous serial protocol interfaces. One of those interfaces that the MPSSE can operate is a SPI bus. By appropriately connecting the FTDI chip pins to a SPI bus peripheral, it is possible to control SPI devices directly from a desktop computer across a USB connection.

To facilitate this, FTDI provides the `libMPSSE-SPI` library. It operates in conjunction with the `D2XX` library (also provided by FTDI) to provide a direct SPI oriented programing interface.

This document describes a Tcl interface to the `libMPSSE-SPI` library. This interface consists, primarily, of “C” code to adapt the calling conventions of `libMPSSE-SPI` to the internal structures of Tcl.

This document is also a literate program. The “C” source code for the Tcl package is also included in the text. The documentation is generated using asciidoc. The source code can be extracted using atangle. The entire set of source files is available from the Tcl-CM3 website.

Readers who may not be practiced at reading literate programs may find it disconcerting that the order is *not* that required by the compiler. Rather, the order of presentation is chosen to facilitate understanding of the program concepts. Tooling is then used to rearrange groups of programming language statements, known as *chunks*, into the order required by the compiler. Rest assured that the required ordering of the program language statements happens correctly as you otherwise could not build the program. In brief, try not to approach reading this document in quite the same way as you might read source code.

Design Concepts

The figure below shows the layering of the software.

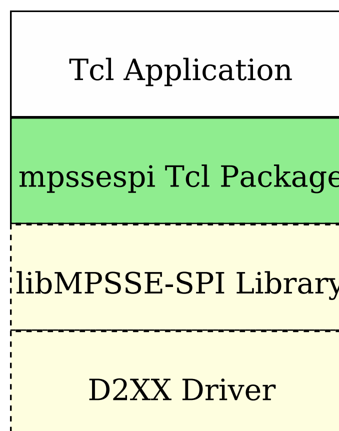

Figure 1: Interface Layers

A Tcl application uses the `mpssespi` package. That package provides an interface to the `libMPSSE-SPI` library. This library is provided by FTDI and operates on top of the `D2XX` driver also provided by FTDI. The default build process of the package staticly links `libMPSSE-SPI` to the Tcl package code so that only the `D2XX` driver must be properly installed on the host system.

Package Interface Considerations

Because the `libMPSSE-SPI` functional interface is oriented to reading and writing data, it might seem natural to design the Tcl interface as a Tcl channel driver. For example, the `fdt2chan` package provides such an interface to the `D2xx` driver. However, a SPI bus has the ability to tranfer data both out and in at the same time and many SPI device operations depend upon that

characteristic of the SPI bus. Reading and writing simultaneously does not fit the semantics of a Tcl file channel. Further, it is possible to interface multiple SPI peripherals to the same FTDI USB to serial converter chip and channel semantics would be further strained to support that concept. So this package provides a rather more direct interface to `libMPSSE-SPI` functions.

The names of the commands provided by the `mpssespi` package were chosen to match those of the “C” functions provided by the library. However, the commands do *not* copy the signature of the library functions exactly. Some of the library functions require configuration data be passed at each invocation. This is tedious, especially considering that much of the configuration information does not change for a given hardware arrangement. Consequently, this package provides a means of storing the necessary configuration information as a set of defaults and commands will use that configuration information where required by the underlying `libMPSSE-SPI` functions. Commands are provided to specify and examine the configuration information in keeping with the usual conventions of Tcl packages for introspection.

In the next section we discuss the data that the `mpssespi` package stores. That is followed by the commands the package provides.

Package Data

This package is written in the *thread neutral* style, *i.e.* this package may be loaded into multiple interpreters simultaneously. Since the FTDI library calls use blocking I/O, some applications may find it necessary to perform the I/O in a separate thread to prevent blocking other activities within the application. Although one would anticipate the I/O blocking time to be very small for SPI bus devices[1], the ability to use threads or separate interpreters is still useful and requires little additional code to implement. To accomplish thread neutrality, package data is held in memory that is associated with the interpreter and *not* as static “C” variables. Tcl provides facilities just for this purpose.

The structure of the package information is given below.

```
<<type definitions>>=
typedef struct MPSSEPkgInfo {
    Tcl_HashTable handleMap ;        /* ❶ */
    unsigned counter ;               /* ❷ */
} MPSSEPkgInfo ;
```

❶ `libMPSSE-SPI` uses a `handle` of its own composition to reference open SPI channels. We will compose strings to represent those handles and use a hash table to map our composed string names to `libMPSSE-SPI` handles.

❷ To compose a package handle name we use a fixed string, `mpssespi`, with an integer number to insure it is unique. This member holds the value of a counter used to derive the unique number for the package handle name.

We need functions to create and delete the package information data.

```
<<utility functions>>=
static ClientData
NewMPSSEPkgInfo(
    Tcl_Interp *interp)
{
    MPSSEPkgInfo *info ;

    info = (MPSSEPkgInfo *)ckalloc(sizeof(*info)) ;
    memset(info, 0, sizeof(*info)) ;
    Tcl_InitHashTable(&info->handleMap, TCL_STRING_KEYS) ;              /* ❶ */
    Tcl_SetAssocData(interp, PACKAGE_NAME, DeleteMPSSEPkgInfo, info) ;/* ❷ */

    return (ClientData)info ;
}
```

[1] Recall that the SPI bus is clocked *synchronously* and its operation does not really depend upon the peripheral device even being present. Consequently, the I/O time is simply the time associated with the number of bits clocked onto and off of the SPI bus.

1. The hash table facilities of Tcl provide the required functionality to map a string to an arbitary piece of data.
2. The allocated data structure is associated with the interpeter by this function.

When the interpreter is deleted, then `DeleteMPSSEPkgInfo()` is invoked. This function is shown below.

```
<<forward utility functions>>=
static void
DeleteMPSSEPkgInfo(
    ClientData clientData,
    Tcl_Interp *interp)
{
    MPSSEPkgInfo *info ;
    Tcl_HashEntry *entry ;
    Tcl_HashSearch searchContext ;

    info = (MPSSEPkgInfo *)clientData ;

    for (entry = Tcl_FirstHashEntry(&info->handleMap, &searchContext) ; /* ❶ */
            entry ; entry = Tcl_NextHashEntry(&searchContext)) {
        MPSSEChanConfig *config = (MPSSEChanConfig *)Tcl_GetHashValue(entry) ;
        ckfree((char *)config) ;
    }
    Tcl_DeleteHashTable(&info->handleMap) ; /* ❷ */

    ckfree((char *)info) ; /* ❸ */
}
```

1. The entries in the hash table are themselves dynamically allocated and so must be freed. Here we iterate through all the hash table entries.
2. We recover the memory for the hash table itself.
3. Finally, the memory for the package information is recovered.

In addition to mapping our own resource names to `libMPSSE-SPI` handles, each open SPI channel has some configuration information. We bundle all of that up into a `MPSSEChanConfig` structure and it is a pointer to this type of object that is the hash table entry. So our resource names actually map to quite a bit of information.

```
<<type definitions>>=
typedef struct MPSSSEChanInfo {
    FT_HANDLE handle ;          /* ❶ */
    bool isInitialized ;        /* ❷ */
    ChannelConfig config ;      /* ❸ */
    uint32 transferOptions ;    /* ❹ */
} MPSSEChanConfig ;
```

1. `handle` is the handle that `libMPSSE-SPI` returns when a SPI channel is opened.
2. `isInitialized` keeps track of whether the channel is initialized. Invoking the `initChannel` command must preceed some other commands.
3. `config` is the channel configuration used by `libMPSSE-SPI`.
4. `transferOptions` are the default options used during read and writes if the caller does not specify any.

As is conventional in Tcl, resources outside of the interpreter are represented as simple strings (*e.g.* file channels) and extension commands then map these arbitrary strings into the extension specific resource information they represent. In the `mpssespi` package, the open SPI channels of `libMPSSE-SPI` are represented by strings of the form `mpssespi<number>`, where `<number>` is replaced by one or more decimal digits. The `NewHandleMapping()` function below creates these handles and sets up the hash table for mapping the handle string names to the channel information.

```
<<utility functions>>=
static Tcl_Obj *
NewHandleMapping(
    MPSSEPkgInfo *info,
    FT_HANDLE handle)
{
    Tcl_Obj *handleName ;
    Tcl_HashEntry *entry ;
    int isNewEntry ;
    MPSSEChanConfig *chanConfig ;

    handleName = Tcl_ObjPrintf("mpssespi%u", info->counter++) ; /* ❶ */

    entry = Tcl_CreateHashEntry(&info->handleMap, Tcl_GetString(handleName),
            &isNewEntry) ; /* ❷ */
    assert(isNewEntry != 0) ;    /* ❸ */

    if (isNewEntry) {
        chanConfig = (MPSSEChanConfig *)ckalloc(sizeof(*chanConfig)) ;/* ❹ */
        chanConfig->handle = handle ;
        chanConfig->isInitialized = 0 ;
        chanConfig->config.ClockRate = DEFAULT_CHANNEL_CONFIG_ClockRate ;
        chanConfig->config.LatencyTimer = DEFAULT_CHANNEL_CONFIG_LatencyTimer ;
        chanConfig->config.configOptions =
                DEFAULT_CHANNEL_CONFIG_configOptions ;
        chanConfig->config.Pin = DEFAULT_CHANNEL_CONFIG_Pin ;
        chanConfig->transferOptions = DEFAULT_TRANSFER_OPTIONS ;
        Tcl_SetHashValue(entry, (ClientData)chanConfig) ;
    } else {
        Tcl_Panic("failed to create mpssespi channel handle entry, \"%s\"\n",
            Tcl_GetString(handleName)) ;
    }

    return handleName ;
}
```

1. Create the handle in an Tcl object.
2. Create the entry in the hash table.
3. Since we are making the hash table keys unique with a counter, we should always create new entries or else something if very wrong. However, we must also account for the fact that assertions go away at release time and `Tcl_Panic()` is used for that purpose.
4. Allocate memory for the channel configuration information and fill in some default values.

Below are the default values of the channel information. Most of the constants are from the `libMPSSE-SPI` header file.

```
<<macro definitions>>=
#define DEFAULT_CHANNEL_CONFIG_ClockRate    1000000
#define DEFAULT_CHANNEL_CONFIG_LatencyTimer 2
#define DEFAULT_CHANNEL_CONFIG_configOptions \
        (SPI_CONFIG_OPTION_MODE0 |\
         SPI_CONFIG_OPTION_CS_DBUS3 |\
         SPI_CONFIG_OPTION_CS_ACTIVELOW)
#define DEFAULT_CHANNEL_CONFIG_Pin          0
```

The default channel configuration values chosen are somewhat arbitrary. The clock rate of 1 MHz is not uncommon for SPI peripherals although many are capable of much faster clock rates. The latency timer value was chosen to work with both older and newer FTDI devices. Mode 0 and active low chip selects are also quite common. It is hard to guess which hardware line is connected to the chip select of the SPI bus peripheral chip. Here we have chosen the first pin available.

```
<<macro definitions>>=
#define DEFAULT_TRANSFER_OPTIONS \
    (SPI_TRANSFER_OPTIONS_SIZE_IN_BYTES |\
     SPI_TRANSFER_OPTIONS_CHIPSELECT_ENABLE|\
     SPI_TRANSFER_OPTIONS_CHIPSELECT_DISABLE)
```

Transfer options can be specified on a per read or write operation basis. To avoid some tedium, we save off default values that will be used if the transfer options are not specified. Again we have chosen values to match the common configuration, of byte oriented interfaces where chip select should be enabled at the beginning of the bus transaction and disabled at the end. The default transfer options can be overidden on a *per* transaction basis for those SPI bus protocols that have different bus protocols. We will see how transfer options are specified below.

Of course if you create something, then you must be prepared to delete it. Resources for channel information are returned by `DeleteHandleMapping()` function which is shown below.

```
<<utility functions>>=
static void
DeleteHandleMapping(
    MPSSEPkgInfo *info,
    Tcl_Obj *handleName)
{
    Tcl_HashEntry *entry ;
    MPSSEChanConfig *chanConfig ;

    entry = Tcl_FindHashEntry(&info->handleMap, Tcl_GetString(handleName)) ;

    assert(entry != NULL) ;
    if (entry) {
        chanConfig = (MPSSEChanConfig *)Tcl_GetHashValue(entry) ;
        ckfree((char *)chanConfig) ;
        Tcl_DeleteHashEntry(entry) ;
    }
}
```

Finally, we need a function to map our “made-up” channel names into the channel information. That work is done by `LookUpHandle()`.

```
<<utility functions>>=
static int
LookUpHandle(
    Tcl_Interp *interp,
    MPSSEPkgInfo *info,
    Tcl_Obj *handleName,
    MPSSEChanConfig **chanConfigPtr)
{
    Tcl_HashEntry *entry ;

    entry = Tcl_FindHashEntry(&info->handleMap, Tcl_GetString(handleName)) ;
    if (entry) {
        if (chanConfigPtr) {  /* ❶ */
            *chanConfigPtr = (MPSSEChanConfig *)Tcl_GetHashValue(entry) ;
        }
        return TCL_OK ;
    } else {
        Tcl_SetObjResult(interp, Tcl_ObjPrintf("unknown channel handle, \"%s\"",
                Tcl_GetString(handleName))) ;
        return TCL_ERROR ;
    }
}
```

1. We do allow for a `NULL` value of the `chanConfigPtr` parameter. This allows a lookup just to verify the channel name. However, this is not a feature of the function used in the package.

Package Commands

With all the preliminaries out of the way, we now turn our attention to the package commands themself. The commands and their names were chose to map directly to the “C” functions provided by `libMPSSE-SPI`. The commands are not a *rote* mapping as we have discussed, particularly in handling configuration information.

Get Number of Channels

One of the first things that an application should determine is the the number of attached FTDI SPI channels. Until a channel is actually open, `libMPSSE-SPI` references them via indices. Those indices start at 0 and run up to the number of attached channels minus one (*i.e.* in traditional “C” array indexing style). The `getNumChannels` command is a direct invocation of the `SPI_GetNumChannels()` library function.

`::mpssespi getNumChannels`
The command returns an integer value that is the the number of SPI channels available.

This is usually the first command that an application would invoke. It is necessary to determine the valid range of channel indices. Note that if there are multiple channels attached, this call still does not tell you enough information to determine to which channel the peripheral is attached. All you can determine here by this command is the total number of attached channels.

```
<<command procedures>>=
static int
GetNumChannelsProc(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *const objv[])
{
    int result = TCL_OK ;
    FT_STATUS status ;
    uint32 numChannels = 0 ;

    if (objc > 1) {
        Tcl_WrongNumArgs(interp, 1, objv, "") ;
        return TCL_ERROR ;
    }

#       ifdef TEST      /* ❶ */
    status = 0 ;
    numChannels = 1 ;
#       else
    status = SPI_GetNumChannels(&numChannels) ;
#       endif /* TEST */
    result = SetStatusResult(interp, status) ;      /* ❷ */
    if (result == TCL_OK) {
        Tcl_SetObjResult(interp, Tcl_NewLongObj(numChannels)) ;
    }
    return result ;
}

<<static data>>=
static char const getNumChannelsCmdName[] = "::mpssespi::getNumChannels" ;
static char const getNumChannelsName[] = "getNumChannels" ;
```

```
<<command creation>>=
Tcl_CreateObjCommand(interp, getNumChannelsCmdName, GetNumChannelsProc,
        clientData, NULL) ;
if (Tcl_Export(interp, ns, getNumChannelsName, 0) != TCL_OK) {
    goto errorout ;
}
if (Tcl_DictObjPut(interp, mapObj, Tcl_NewStringObj(getNumChannelsName, -1),
        Tcl_NewStringObj(getNumChannelsCmdName, -1)) != TCL_OK) {
    goto errorout ;
}
```

1. Support for testing the package code without a FTDI device installed.
2. Translating `libMPSSE-SPI` returned status values to meaningful Tcl interpreter results is accomplished in one place below.

We will follow the pattern established above for the commands. First we present the source code to the command followed by the statements to create the command. For this package, we export all the commands and will then create a `namespace ensemble` command that has the same name as the namespace where the commands reside. We will also supply an ensemble mapping dictionary. The intent of all of this is to allow the ensemble of the package to be extended at the script level.

We will also establish a pattern of including `tcltest` test code along with the package implementation. Testing an attached peripheral device poses special challenges. We must make assumptions about what is connected. Replicating the testing environment may be difficult for others. So to overcome some of these difficulties, we define a conditional compilation symbol, `TEST`. If this symbol is defined to the preprocessor, the actual calls to the `libMPSSE-SPI` library are replaced by stubbed out results. This allows testing of the other logic without having an actual FTDI device in place.

```
<<getNumChannels tests>>=
test getNumChannels-1.0 {
    getNumChannels test
} -setup {
} -cleanup {
} -body {
    ::mpssespi getNumChannels
} -result {1}
```

Get Channel Info

Given a channel index number, the `getChannelInfo` command returns some information about the channel. The `getChannelInfo` command is a direct invocation of the `SPI_GetChannelInfo()` library function.

`::mpssespi getChannelInfo` *?chanindex?*

chanindex
: A `libMPSSE-SPI` channel index. This is an integer value between 0 and the number of channels returned by the `getNumChannels` command minus 1. If the *chanindex* argument is missing, then it is assumed to have the value of 0.

The command returns a dictionary containing the channel information. The keys of the dictionary and the meaning of the corresponding value are:

opened
: A boolean value indicating if the channel is open

hispeed
: A boolean value indicating whether the channel is a high speed channel.

type
: An numeric value representing the device type.

idvendor
: A numeric value representing the device vendor identifier.

idproduct
: A numeric value representing the device product identifier.

locid
: A numeric value representing the location identifier.

serialnumber
: A string value giving the serial number of the channel.

description
: A string value giving a description of the channel.

The information given by this command can be used to determine which SPI channel is connected to a particular peripheral. By some experimentation of hot plugging and unplugging the FTDI converter, you should be able to determine the serial number or location ID of the channel attached to your peripheral. The details of this are, of course, specific to your hardware arrangement but once determined it is a simple script to search the channel information of all the attached SPI channels for the one connected to your peripheral. It is also quite common to have only a single FTDI converter attached. In this case, finding the correct channel is trivial. The reason all this is necessary is that we must know the index of the channel we wish to open and those indices are not predetermined.

```
<<command procedures>>=
static int
GetChannelInfoProc(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *const objv[])
{
    int result ;
    uint32 chanIndex ;
    FT_DEVICE_LIST_INFO_NODE deviceInfo ;
    FT_STATUS status ;
    Tcl_Obj *infoObj ;

    /* ❶ */
    if (objc == 1) {
        chanIndex = 0 ;
    } else if (objc == 2) {
```

```
        int chanIndexObjValue ;

        if (Tcl_GetIntFromObj(interp, objv[1], &chanIndexObjValue) != TCL_OK) {
            return TCL_ERROR ;
        }
        chanIndex = chanIndexObjValue ;
    } else {
        Tcl_WrongNumArgs(interp, 1, objv, "?channel?") ;
        return TCL_ERROR ;
    }

    /* ❷ */
#       ifdef TEST
    (void)chanIndex ;
    status = 0 ;
    deviceInfo.Flags = 0 ;
    deviceInfo.Type = 0x41 ;
    deviceInfo.ID = 0x42 ;
    deviceInfo.LocId = 0x43 ;
    strcpy(deviceInfo.SerialNumber, "001") ;
    strcpy(deviceInfo.Description, "This is a test") ;
    deviceInfo.ftHandle = (void *)0xfffffe ;
#       else
    status = SPI_GetChannelInfo(chanIndex, &deviceInfo) ;
#       endif /* TEST */
    result = SetStatusResult(interp, status) ;

    /* ❸ */
    if (result == TCL_OK) {
        infoObj = Tcl_NewDictObj() ;

        result = Tcl_DictObjPut(interp, infoObj,
                Tcl_NewStringObj("opened", -1),
                Tcl_NewBooleanObj((deviceInfo.Flags & FT_FLAGS_OPENED) != 0)) ;
        if (result != TCL_OK) {
            goto errout ;       /* ❹ */
        }

        result = Tcl_DictObjPut(interp, infoObj,
                Tcl_NewStringObj("hispeed", -1),
                Tcl_NewBooleanObj((deviceInfo.Flags & FT_FLAGS_HISPEED) != 0)) ;
        if (result != TCL_OK) {
            goto errout ;
        }

        result = Tcl_DictObjPut(interp, infoObj,
                Tcl_NewStringObj("type", -1), Tcl_NewIntObj(deviceInfo.Type)) ;
        if (result != TCL_OK) {
            goto errout ;
        }

        result = Tcl_DictObjPut(interp, infoObj,            /* ❺ */
                Tcl_NewStringObj("idvendor", -1),
                Tcl_NewLongObj(EXTRACT_BITFIELD(deviceInfo.ID, 16, 16))) ;
        if (result != TCL_OK) {
            goto errout ;
        }

        result = Tcl_DictObjPut(interp, infoObj,
                Tcl_NewStringObj("idproduct", -1),
                Tcl_NewLongObj(EXTRACT_BITFIELD(deviceInfo.ID, 0, 16))) ;
        if (result != TCL_OK) {
```

```
            goto errout ;
        }

        result = Tcl_DictObjPut(interp, infoObj,
                Tcl_NewStringObj("locid", -1), Tcl_NewIntObj(deviceInfo.LocId)) ;
        if (result != TCL_OK) {
            goto errout ;
        }

        result = Tcl_DictObjPut(interp, infoObj,
                Tcl_NewStringObj("serialnumber", -1),
                Tcl_NewStringObj(deviceInfo.SerialNumber, -1)) ;
        if (result != TCL_OK) {
            goto errout ;
        }

        result = Tcl_DictObjPut(interp, infoObj,
                Tcl_NewStringObj("description", -1),
                Tcl_NewStringObj(deviceInfo.Description, -1)) ;
        if (result != TCL_OK) {
            goto errout ;
        }

        Tcl_SetObjResult(interp, infoObj) ;
    }

    return result ;

errout:
    Tcl_DecrRefCount(infoObj) ;
    return TCL_ERROR ;
}

<<static data>>=
static char const getChannelInfoCmdName[] = "::mpssespi::getChannelInfo" ;
static char const getChannelInfoName[] = "getChannelInfo" ;

<<command creation>>=
Tcl_CreateObjCommand(interp, getChannelInfoCmdName, GetChannelInfoProc,
        clientData, NULL) ;
if (Tcl_Export(interp, ns, getChannelInfoName, 0) != TCL_OK) {
    goto errorout ;
}
if (Tcl_DictObjPut(interp, mapObj, Tcl_NewStringObj(getChannelInfoName, -1),
        Tcl_NewStringObj(getChannelInfoCmdName, -1)) != TCL_OK) {
    goto errorout ;
}
```

1. Argument parsing and processing.
2. Invoke `SPI_GetChannelInfo()`.
3. Convert the returned values into a Tcl dictionary.
4. Okay, for all you `goto` whiners out there. Yes, there will be `goto`s in the code. But note, they always jump **forward** in the code (never backward as that is truly just wrong) and they are used to handle error exceptions which is not a language concept that "C" supports. And yes, the code could have been constructed as some deeply nested `if, then` construct or divided into tiny functions or some other equally contrived construct. But when you have long sequences of calls as we do here, the `goto` avoids the strain of dealing with an excessive nesting depth or the lexical distance that small functions impose. So get over it! No doubt that you will also find other conventions of my coding style that seem quirky to you (*e.g.* adding spaces around the `;` statement terminator). I can suggest the `indent` program for you.

5 We divide up the returned `id` into its vendor and product portions.

The code for this command also follows a pattern that we will see again. Since the underlying libary call simply produces information, most of the code is devoted to converting that information from the form it is returned by the library to Tcl objects which are gathered into a dictionary.

```
<<getChannelInfo tests>>=
test getChannelInfo-1.0 {
    getChannelInfo test
} -setup {
} -cleanup {
} -body {
    set info [::mpssespi getChannelInfo 0]
    dict get $info opened
} -result {0}
```

Open Channel

The first step in being able to use a `libMPSSE-SPI` channel for SPI control is to open it. Opening the channel associates a handle to a channel index and prepares the channel for subsequent operations. The `openChannel` command invokes the `SPI_OpenChannel()` function.

`::mpssespi openChannel` *?chanindex?*

chanindex
A `libMPSSE-SPI` channel index. This is an integer value between 0 and the number of channels returned by the `getNumChannels` command minus 1. If the *chanindex* argument is missing, then it is assumed to have the value of 0.

The `openChannel` command opens a SPI channel and returns a channel handle that is to be passed as an argument to other `mpssespi` package commands that operate on SPI channels. The returned handle is a string, but its value is only meaningful to the `mpssespi` package and its form should not be relied upon by the caller. However, the caller may assume that the returned string is unique among all open SPI bus channels (*i.e.* the returned handle is suitable to use as a Tcl array index).

```
<<command procedures>>=
static int
OpenChannelProc(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *const objv[])
{
    uint32 chanIndex ;
    FT_STATUS status ;
    FT_HANDLE handle ;
    int result ;

    if (objc == 1) {
        chanIndex = 0 ;
    } else if (objc == 2) {
        int chanIndexObjValue ;

        if (Tcl_GetIntFromObj(interp, objv[1], &chanIndexObjValue) != TCL_OK) {
            return TCL_ERROR ;
        }
        chanIndex = chanIndexObjValue ;
    } else {
```

```
        Tcl_WrongNumArgs(interp, 1, objv, "?channel?") ;
        return TCL_ERROR ;
    }

#       ifdef TEST
    (void)chanIndex ;
    status = 0 ;
    handle = (void *)0xfffffe ;
#       else
    status = SPI_OpenChannel(chanIndex, &handle) ;
#       endif /* TEST */
    result = SetStatusResult(interp, status) ;

    if (result == TCL_OK) {
        Tcl_Obj *handleName = NewHandleMapping((MPSSEPkgInfo *)clientData,
                handle) ;        /* ❶ */
        Tcl_SetObjResult(interp, handleName) ;
    }

    return result ;
}

<<static data>>=
static char const openChannelCmdName[] = "::mpssespi::openChannel" ;
static char const openChannelName[] = "openChannel" ;

<<command creation>>=
Tcl_CreateObjCommand(interp, openChannelCmdName, OpenChannelProc,
        clientData, NULL) ;
if (Tcl_Export(interp, ns, openChannelName, 0) != TCL_OK) {
    goto errorout ;
}
if (Tcl_DictObjPut(interp, mapObj, Tcl_NewStringObj(openChannelName, -1),
        Tcl_NewStringObj(openChannelCmdName, -1)) != TCL_OK) {
    goto errorout ;
}
```

❶ We finally see a channel information function invocation. Note that the `clientData` argument to the command function is being cast to a pointer to a `mpssespi` package specific object. The command creation code arranges for that value to be passed to each of the commands in the package. We will see how that happens below, but for now simply understand that some package associated data was allocated and is made available to the package commands via the `clientData` argument.

```
<<openChannel tests>>=
test openChannel-1.0 {
    openChannel test
} -setup {
} -cleanup {
    ::mpssespi closeChannel $handle
} -body {
    set handle [::mpssespi openChannel 0]
} -result {mpsse*} -match glob
```

Init Channel

The second step to using a SPI channel is to initialize a previously opened channel. It turns out that `libMPSSE-SPI` separates the operations of opening a channel and initializing that channel into separate functions. This separation adds an extra command to the set up but does allow for channels to be re-initialized without being closed. The main function of `SPI_InitChannel()`

seems to be to set up the configuration for the channel. It turns out that there are alot of configuration options for a SPI channel. So we have created some configuration commands to deal with reading and updating the channel configuration.

So after opening a channel it is necessary to initialize it. If you wish to change the configuration used for the initialization, then you can set different configuration values **before** invoking the `initChannel` command. Otherwise, you end up with the default configuration.

`::mpssespi initChannel` *?handle?*

handle
A `mpssespi` channel handle as returned from a successful call to the `openChannel` command.

The `initChannel` command initializes an open SPI channel. Configuration information associated with the channel is used for the initialization. The command returns the empty string.

```
<<command procedures>>=
static int
InitChannelProc(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *const objv[])
{
    MPSSEPkgInfo *pkgInfo = (MPSSEPkgInfo *)clientData ;
    Tcl_Obj *handleObj ;
    MPSSEChanConfig *chanConfig ;
    FT_STATUS status ;
    int result ;

    if (objc == 2) {
        handleObj = objv[1] ;
    } else {
        Tcl_WrongNumArgs(interp, 1, objv, "channelHandle") ;
        return TCL_ERROR ;
    }

    /* ❶*/
    if (LookUpHandle(interp, pkgInfo, handleObj, &chanConfig) != TCL_OK) {
        return TCL_ERROR ;
    }
#       ifdef TEST
    status = 0 ;
#       else
    status = SPI_InitChannel(chanConfig->handle, &chanConfig->config) ; /* ❷ */
#       endif /* TEST */
    result = SetStatusResult(interp, status) ;
    if (result == TCL_OK) {
        chanConfig->isInitialized = 1 ;      /* ❸ */
        Tcl_ResetResult(interp) ;
    }

    return result ;
}

<<static data>>=
static char const initChannelCmdName[] = "::mpssespi::initChannel" ;
static char const initChannelName[] = "initChannel" ;

<<command creation>>=
Tcl_CreateObjCommand(interp, initChannelCmdName, InitChannelProc,
        clientData, NULL) ;
```

```
if (Tcl_Export(interp, ns, initChannelName, 0) != TCL_OK) {
    goto errorout ;
}
if (Tcl_DictObjPut(interp, mapObj, Tcl_NewStringObj(initChannelName, -1),
        Tcl_NewStringObj(initChannelCmdName, -1)) != TCL_OK) {
    goto errorout ;
}
```

1. Here we see how the package information (as passed via the `clientData` argument) is used to map the channel handle back to information that is needed to invoke `libMPSSE-SPI` functions. We will see this pattern often in the other commands.
2. Note that initializing the channel does not use all of the configuration information that we are holding. Only that part directly pertaining to the channel is handled by initialization. The *transfer options* are used on a per SPI bus transaction basis and we will see them used later.
3. We now mark the channel as initialized. The reason for tracking the initialization is that some functions on an open channel will “hang” if they are invoked before the channel is initialized. Tracking the state just prevents small programmer errors from becoming a major inconvenience.

```
<<initChannel tests>>=
test initChannel-1.0 {
    initialize an open channel
} -setup {
    set handle [::mpssespi openChannel 0]
} -cleanup {
    ::mpssespi closeChannel $handle
} -body {
    ::mpssespi initChannel $handle
} -result {}
```

Get Channel Configuration

We can no long postpone dealing with channel configuration. There are two main pieces of configuration information. One part deals with configuring the channel as is done in `SPI_InitChannel()` and the other part is used for each SPI bus transaction.

We first present the `getChannelConfig` command. This allows us to inspect the configuration that is being stored.

`::mpssespi getChannelConfig` *handle*

handle
: A `mpssespi` channel handle as returned from a successful call to the `openChannel` command.

The `getChannelConfig` command returns a dictionary containing the configuration information of the open channel designated by *handle*. The keys of the dictionary and an explanation of the meaning of the corresponding value is given below.

clockrate
: The SPI bus clock rate. Valid values are from 0 to 30000000 Hz (30 MHz)

latencytimer
: The latency timer value. Valid values are from 0 to 255. FTDI suggests minimum values of 1 for lower speed chips and 2 for higher speed chips.

mode
: The SPI bus mode. This is the standard SPI bus mode designation that controls when data is latched relative to the clock edges. Valid values are 0 through 3.

csline
: The bus line that is to be used as the SPI bus chip select. Valid values are strings from the set: "DBUS3", "DBUS4", "DBUS5", "DBUS6" or "DBUS7". These names refer to pin designation on FTDI chips.

csactive
: The level at which the chip select line is active. Valid values are strings from the set: "high" or "low".

initpindir
: The I/O pin direction to be established when the channel is initialized. The value of this key is also a dictionary with keys "0" - "7" and values of "in" or "out".

initpinvalue
: The value settings for the I/O pins that are established when the channel is initialized. The value of this key is also a dictionary with keys "0" - "7" and values that are any value Tcl representation of a boolean. A "true" value requests the pin be set high and a "false" value requests the pin be set low.

closepindir
: The I/O pin direction to be established when the channel is closed. The value of this key is also a dictionary with the same keys as for the `initpindir` key.

closepinvalue
: The value settings for the I/O pins that are established when the channel is closed. The value of this key is also a dictionary with the same keys as for the `initpinvalue` key.

xferunits
: Specifies the units for which SPI bus data transfers are to happen. Valid values are "bits" or "bytes".

csenable
: A boolean that specifies whether the chip select line is to be made active before the SPI bus transaction.

csdisable
: A boolean that specifies whether the chip select line is to be made inactive after the SPI bus transaction.

The implementation of `getChannelConfig` is conceptually quite simple. We use the handle to look up the channel information and then build a Tcl dictionary object to hold that information. The code is long and rather tedious since there are so many dictionary keys to handle.

```
<<command procedures>>=
static int
GetChannelConfigProc(
    ClientData clientData,
```

```
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *const objv[])
{
    MPSSEPkgInfo *pkgInfo = (MPSSEPkgInfo *)clientData ;
    Tcl_Obj *handleObj ;
    MPSSEChanConfig *chanConfig ;
    Tcl_Obj *configObj ;
    char *csline ;
    Tcl_Obj *pinObj ;

    if (objc == 2) {
        handleObj = objv[1] ;
    } else {
        Tcl_WrongNumArgs(interp, 1, objv, "channelHandle") ;
        return TCL_ERROR ;
    }

    if (LookUpHandle(interp, pkgInfo, handleObj, &chanConfig) != TCL_OK) {
        return TCL_ERROR ;
    }

    configObj = Tcl_NewDictObj() ;
    /*
     * clockrate
     */
    if (Tcl_DictObjPut(interp, configObj, Tcl_NewStringObj("clockrate", -1),
            Tcl_NewLongObj(chanConfig->config.ClockRate)) != TCL_OK) {
        goto errorout ;
    }
    /*
     * latencytimer
     */
    if (Tcl_DictObjPut(interp, configObj, Tcl_NewStringObj("latencytimer", -1),
            Tcl_NewIntObj(chanConfig->config.LatencyTimer)) != TCL_OK) {
        goto errorout ;
    }
    /*
     * mode
     */
    if (Tcl_DictObjPut(interp, configObj, Tcl_NewStringObj("mode", -1),
            Tcl_NewIntObj(
                chanConfig->config.configOptions & SPI_CONFIG_OPTION_MODE_MASK))
            != TCL_OK) {
        goto errorout ;
    }
    /*
     * csline
     */
    switch (chanConfig->config.configOptions & SPI_CONFIG_OPTION_CS_MASK) {
    case SPI_CONFIG_OPTION_CS_DBUS3:
        csline = "DBUS3" ;
        break ;

    case SPI_CONFIG_OPTION_CS_DBUS4:
        csline = "DBUS4" ;
        break ;

    case SPI_CONFIG_OPTION_CS_DBUS5:
        csline = "DBUS5" ;
        break ;
```

```
    case SPI_CONFIG_OPTION_CS_DBUS6:
        csline = "DBUS6" ;
        break ;

    case SPI_CONFIG_OPTION_CS_DBUS7:
        csline = "DBUS7" ;
        break ;

    default:
        csline = "UNKNOWN" ;
        break ;
    }
    if (Tcl_DictObjPut(interp, configObj, Tcl_NewStringObj("csline", -1),
            Tcl_NewStringObj(csline, -1)) != TCL_OK) {
        goto errorout ;
    }
    /*
     * csactive
     */
    if (Tcl_DictObjPut(interp, configObj, Tcl_NewStringObj("csactive", -1),
            Tcl_NewStringObj(
            (chanConfig->config.configOptions & SPI_CONFIG_OPTION_CS_ACTIVELOW) ?
                "low" : "high", -1)) != TCL_OK) {
        goto errorout ;
    }
    /*
     * initpindir
     */
    if (GetDirectionDict(interp,
            EXTRACT_BITFIELD(chanConfig->config.Pin, PIN_INITIAL_DIR_OFFSET,
            PIN_BITFIELD_LENGTH), &pinObj) != TCL_OK) { /* ❶ */
        goto errorout ;
    }
    if (Tcl_DictObjPut(interp, configObj, Tcl_NewStringObj("initpindir", -1),
            pinObj) != TCL_OK) {
        goto pinerrorout ;
    }
    /*
     * initpinvalue
     */
    if (GetPinValueDict(interp,
            EXTRACT_BITFIELD(chanConfig->config.Pin, PIN_INITIAL_VALUE_OFFSET,
            PIN_BITFIELD_LENGTH), &pinObj) != TCL_OK) { /* ❷ */
        goto errorout ;
    }
    if (Tcl_DictObjPut(interp, configObj, Tcl_NewStringObj("initpinvalue", -1),
            pinObj) != TCL_OK) {
        goto pinerrorout ;
    }
    /*
     * closepindir
     */
    if (GetDirectionDict(interp,
            EXTRACT_BITFIELD(chanConfig->config.Pin, PIN_CLOSE_DIR_OFFSET,
            PIN_BITFIELD_LENGTH), &pinObj) != TCL_OK) {
        goto errorout ;
    }
    if (Tcl_DictObjPut(interp, configObj, Tcl_NewStringObj("closepindir", -1),
            pinObj) != TCL_OK) {
        goto pinerrorout ;
    }
    /*
```

```
     * closepinvalue
     */
    if (GetPinValueDict(interp,
            EXTRACT_BITFIELD(chanConfig->config.Pin, PIN_CLOSE_VALUE_OFFSET,
            PIN_BITFIELD_LENGTH), &pinObj) != TCL_OK) {
        goto errorout ;
    }
    if (Tcl_DictObjPut(interp, configObj, Tcl_NewStringObj("closepinvalue", -1),
            pinObj) != TCL_OK) {
        goto pinerrorout ;
    }
    /*
     * xferunits
     */
    if (Tcl_DictObjPut(interp, configObj, Tcl_NewStringObj("xferunits", -1),
            Tcl_NewStringObj(IS_BIT_XFER(chanConfig->transferOptions) ?
                "bits" : "bytes", -1)) != TCL_OK) { /* ❸ */
        goto errorout ;
    }
    /*
     * csenable
     */
    if (Tcl_DictObjPut(interp, configObj, Tcl_NewStringObj("csenable", -1),
            Tcl_NewBooleanObj((chanConfig->transferOptions &
            SPI_TRANSFER_OPTIONS_CHIPSELECT_ENABLE) != 0)) != TCL_OK) {
        goto errorout ;
    }
    /*
     * csdisable
     */
    if (Tcl_DictObjPut(interp, configObj, Tcl_NewStringObj("csdisable", -1),
            Tcl_NewBooleanObj((chanConfig->transferOptions &
            SPI_TRANSFER_OPTIONS_CHIPSELECT_DISABLE) != 0)) != TCL_OK) {
        goto errorout ;
    }

    Tcl_SetObjResult(interp, configObj) ;
    return TCL_OK ;

pinerrorout :
    Tcl_DecrRefCount(pinObj) ;

errorout:
    Tcl_DecrRefCount(configObj) ;
    return TCL_ERROR ;
}

<<static data>>=
static char const getChannelConfigCmdName[] = "::mpssespi::getChannelConfig" ;
static char const getChannelConfigName[] = "getChannelConfig" ;

<<command creation>>=
Tcl_CreateObjCommand(interp, getChannelConfigCmdName, GetChannelConfigProc,
        clientData, NULL) ;
if (Tcl_Export(interp, ns, getChannelConfigName, 0) != TCL_OK) {
    goto errorout ;
}
if (Tcl_DictObjPut(interp, mapObj, Tcl_NewStringObj(getChannelConfigName, -1),
        Tcl_NewStringObj(getChannelConfigCmdName, -1)) != TCL_OK) {
    goto errorout ;
}
```

1. `GetDirectionDict()` is discussed below.
2. `GetPinValueDict()` is also discussed below.
3. Constants of the form `SPI_TRANSFER_OPTIONS_XXX` are found in the `libMPSSE_spi.h` header file.

```
<<getChannelConfig tests>>=
test getChannelConfig-1.0 {
    read the channel configuration
} -setup {
    set handle [::mpssespi openChannel 0]
} -cleanup {
    ::mpssespi closeChannel $handle
} -body {
    dict get [::mpssespi getChannelConfig $handle] clockrate
} -result {10000000}
```

Pin Direction Configuration

The direction of the I/O pins is specified as a Tcl dictionary. The function below converts the bit encoded values of the configuration information into a Tcl dictionary. Later, we will need the complement of this function to go the other way.

The code involves some bit-twiddling and we will see a lot more of that since `libMPSSE-SPI` encodes several portions of the configuration information as bit fields. The strategy here is to use a `mask` variable, which is shifted left as we proceed along the bitfield, to probe for set bits within the bit field.

```
<<utility functions>>=
static int
GetDirectionDict(
    Tcl_Interp *interp,
    uint8 dirBitMask,
    Tcl_Obj **dirDictObjPtr)
{
    char key[2] ; /* ❶ */
    Tcl_Obj *dirObj ;
    uint8 mask ;
    int count ;

    assert(dirDictObjPtr != NULL) ;

    dirObj = Tcl_NewDictObj() ;

    key[0] = '0' ;
    key[1] = '\0' ;
    for (mask = 1, count = PIN_BITFIELD_LENGTH ; count > 0 ;
            mask <<= 1, count--) {
        if (Tcl_DictObjPut(interp, dirObj, Tcl_NewStringObj(key, -1),
                Tcl_NewStringObj((dirBitMask & mask) ? "out" : "in", -1))
                != TCL_OK) {
            goto errorout ;
        }

        key[0]++ ; /* ❷ */
    }

    *dirDictObjPtr = dirObj ;
    return TCL_OK ;

errorout:
    Tcl_DecrRefCount(dirObj) ;
    *dirDictObjPtr = NULL ;
```

```
    return TCL_ERROR ;
}
```

1. Since we are putting things in a dictionary, we will need to compose a string for the dictionary key.
2. Note the assumption of a character encoding where the next decimal digit is also the next integer value. This works of ASCII, of course, and is just too tempting not to use. A better way would be to use a static array of characters of the decimal digits and iterate through that array to fill in the key string value. But let's not guild the lily.

We also need a similar function to deal with the pin value configuration information. It follows the same pattern shown above, producing a dictionary of boolean values.

```
<<utility functions>>=
static int
GetPinValueDict(
    Tcl_Interp *interp,
    uint8 pinBitMask,
    Tcl_Obj **pinDictObjPtr)
{
    char key[2] ;
    Tcl_Obj *pinObj ;
    uint8 mask ;
    int count ;

    assert(pinDictObjPtr != NULL) ;

    pinObj = Tcl_NewDictObj() ;

    key[0] = '0' ;
    key[1] = '\0' ;
    for (mask = 1, count = PIN_BITFIELD_LENGTH  ; count > 0 ;
            mask <<= 1, count--) {
        if (Tcl_DictObjPut(interp, pinObj, Tcl_NewStringObj(key, -1),
                Tcl_NewBooleanObj((pinBitMask & mask) ? 1 : 0))
                != TCL_OK) {
            goto errorout ;
        }

        key[0]++ ;
    }

    *pinDictObjPtr = pinObj ;
    return TCL_OK ;

errorout:
    Tcl_DecrRefCount(pinObj) ;
    *pinDictObjPtr = NULL ;
    return TCL_ERROR ;
}
```

Set Channel Configuration

We also need a command for setting the channel configuration. Note that changing the channel configuration with this command does not cause that configuration to be updated to the FTDI chip. That is only done when the `initChannel` command is invoked. It is just the way the library works. So the usual sequence is to open the channel, modify the configuration information from its default values if necessary and then initialize the channel. So, the command only modifies the configuration information that is associated with an open channel. It is still necessary to invoke `initChannel` to set the new values of the configuration into the chip.

`::mpssespi setChannelConfig` *handle configuration*

handle
A `mpssespi` channel handle as returned from a successful call to the `openChannel` command.

configuration
A dictionary value that contains the channel configuration information.

The `setChannelConfig` command changes the channel configuration information. This command only changes those parts of the channel configuration given by the `configuration` dictionary value. Configuration information for missing keys is *not* changed. Dictionary keys that do not match one of the values below are *silently ignored*.

The keys of the dictionary are the same as those returned from the `getChannelConfig` command. They are listed below for easy reference. The command returns the empty string.

clockrate
The SPI bus clock rate. Valid values are from 0 to 30000000 Hz (30 MHz)

latencytimer
The latency timer value. Valid values are from 0 to 255. FTDI suggests minimum values of 1 for lower speed chips and 2 for higher speed chips.

mode
The SPI bus mode. This is the standard SPI bus mode designation that controls when data is latched relative to the clock edges. Valid values are 0 through 3.

csline
The bus line that is to be used as the SPI bus chip select. Valid values are strings from the set: "DBUS3", "DBUS4", "DBUS5", "DBUS6" or "DBUS7".

csactive
The level at which the chip select line is active. Valid values are strings from the set: "high" or "low".

initpindir
The I/O pin direction to be established when the channel is initialized. The value of this key is also a dictionary with keys "0" - "7" and values of "in" or "out".

initpinvalue
The value settings for the I/O pins that are established when the channel is initialized. The value of this key is also a dictionary with keys "0" - "7" and values that are any value Tcl representation of a boolean. A "true" value request. the pin be set high and a "false" value requests the pin be set low.

closepindir
The I/O pin direction to be established when the channel is closed. The value of this key is also a dictionary with the same keys as for the `initpindir` key.

closepinvalue
The value settings for the I/O pins that are established when the channel is closed. The value of this key is also a dictionary with the same keys as for the `initpinvalue` key.

xferunits
Specifies the units for which SPI bus data transfers are to happen. Valid values are "bits" or "bytes".

csenable
A boolean that specifies whether the chip select line is to be made active before the SPI bus transaction.

csdisable
A boolean that specifies whether the chip select line is to be made inactive after the SPI bus transaction.

The implementation strategy here is simlar to that of the `getChannelConfiguration` command. Again the code is rather

long and tedious to deal with all the different configuration settings. However, one consideration is to make the setting of the configuration transactional. So we only change the configuration if there is no error in setting any of the fields. We accomplish that by using a working copy of the channel configuration while we are decoding the `configuration` argument and then, if all went correctly, overwriting the channel configuration with the modified information.

```
<<command procedures>>=
static int
SetChannelConfigProc(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *const objv[])
{
    MPSSEPkgInfo *pkgInfo = (MPSSEPkgInfo *)clientData ;
    MPSSEChanConfig *chanConfig ;
    MPSSEChanConfig newConfig ;
    Tcl_Obj *configObj ;
    Tcl_Obj *keyObj ;
    Tcl_Obj *valueObj ;

    if (objc != 3) {
        Tcl_WrongNumArgs(interp, 1, objv, "channelHandle configuration") ;
        return TCL_ERROR ;
    }

    if (LookUpHandle(interp, pkgInfo, objv[1], &chanConfig) != TCL_OK) {
        return TCL_ERROR ;
    }
    newConfig = *chanConfig ;       /* ❶ */

    configObj = objv[2] ;
    /*
     * clockrate
     */
    keyObj = Tcl_NewStringObj("clockrate", -1) ;
    if (Tcl_DictObjGet(interp, configObj, keyObj, &valueObj) != TCL_OK) {
        goto errorout ;
    }
    if (valueObj) {      /* ❷ */
        long clockrate ;

        if (Tcl_GetLongFromObj(interp, valueObj, &clockrate) != TCL_OK) {
            goto errorout ;
        }
        if (clockrate < 0 || clockrate > 30000000) {
            Tcl_SetObjResult(interp, Tcl_ObjPrintf(
                "bad clockrate value, %ld: should be between 0 and 30000000",
                clockrate)) ;
            goto errorout ;
        }
        newConfig.config.ClockRate = clockrate ;
    }
    /*
     * latencytimer
     */
    Tcl_SetStringObj(keyObj, "latencytimer", -1) ;
    if (Tcl_DictObjGet(interp, configObj, keyObj, &valueObj) != TCL_OK) {
        goto errorout ;
    }
    if (valueObj) {
        int latencytimer ;
```

```
        if (Tcl_GetIntFromObj(interp, valueObj, &latencytimer) != TCL_OK) {
            goto errorout ;
        }
        if (latencytimer < 0 || latencytimer > 255) {
            Tcl_SetObjResult(interp, Tcl_ObjPrintf(
                "bad latency timer value, %d: should be between 0 and 255",
                latencytimer)) ;
            goto errorout ;
        }
        newConfig.config.LatencyTimer = latencytimer ;
    }
    /*
     * mode
     */
    Tcl_SetStringObj(keyObj, "mode", -1) ;
    if (Tcl_DictObjGet(interp, configObj, keyObj, &valueObj) != TCL_OK) {
        goto errorout ;
    }
    if (valueObj) {
        int mode ;

        if (Tcl_GetIntFromObj(interp, valueObj, &mode) != TCL_OK) {
            goto errorout ;
        }
        if (mode < 0 || mode > 3) {
            Tcl_SetObjResult(interp, Tcl_ObjPrintf(
                "bad mode value, %d: should be between 0 and 3", mode)) ;
            goto errorout ;
        }
        newConfig.config.configOptions =
            INSERT_BITFIELD(newConfig.config.configOptions, mode, 0, 2) ;
    }
    /*
     * csline
     */
    Tcl_SetStringObj(keyObj, "csline", -1) ;
    if (Tcl_DictObjGet(interp, configObj, keyObj, &valueObj) != TCL_OK) {
        goto errorout ;
    }
    if (valueObj) {
        char const *csline ;
        uint8 csconfig ;

        csline = Tcl_GetString(valueObj) ;
        if (strcmp(csline, "DBUS3") == 0) {
            csconfig = SPI_CONFIG_OPTION_CS_DBUS3 ;
        } else if (strcmp(csline, "DBUS4") == 0) {
            csconfig = SPI_CONFIG_OPTION_CS_DBUS4 ;
        } else if (strcmp(csline, "DBUS5") == 0) {
            csconfig = SPI_CONFIG_OPTION_CS_DBUS5 ;
        } else if (strcmp(csline, "DBUS6") == 0) {
            csconfig = SPI_CONFIG_OPTION_CS_DBUS6 ;
        } else if (strcmp(csline, "DBUS7") == 0) {
            csconfig = SPI_CONFIG_OPTION_CS_DBUS7 ;
        } else {
            Tcl_SetObjResult(interp, Tcl_ObjPrintf(
    "bad csline value, %s: should be one of DBUS3, DBUS4, DBUS5, DBUS6, DBUS7",
            csline)) ;
            goto errorout ;
        }

        newConfig.config.configOptions =
```

```
            CLEARFIELD(newConfig.config.configOptions, 2, 3) | csconfig ;
    }
    /*
     * csactive
     */
    Tcl_SetStringObj(keyObj, "csactive", -1) ;
    if (Tcl_DictObjGet(interp, configObj, keyObj, &valueObj) != TCL_OK) {
        goto errorout ;
    }
    if (valueObj) {
        char const *csactive ;

        csactive = Tcl_GetString(valueObj) ;
        if (strcmp(csactive, "high") == 0) {
            newConfig.config.configOptions &= ~SPI_CONFIG_OPTION_CS_ACTIVELOW ;
        } else if (strcmp(csactive, "low") == 0) {
            newConfig.config.configOptions |= SPI_CONFIG_OPTION_CS_ACTIVELOW ;
        } else {
            Tcl_SetObjResult(interp, Tcl_ObjPrintf(
                    "bad csactive value, %s: should be high or low",
                    csactive)) ;
            goto errorout ;
        }
    }
    /*
     * initpindir
     */
    Tcl_SetStringObj(keyObj, "initpindir", -1) ;
    if (Tcl_DictObjGet(interp, configObj, keyObj, &valueObj) != TCL_OK) {
        goto errorout ;
    }
    if (valueObj) {
        uint8 initpindir = EXTRACT_BITFIELD(newConfig.config.Pin,
            PIN_INITIAL_DIR_OFFSET, PIN_BITFIELD_LENGTH) ;

        if (GetDirectionBitMask(interp, valueObj, &initpindir) != TCL_OK) {
            goto errorout ;
        }
        newConfig.config.Pin = INSERT_BITFIELD(newConfig.config.Pin,
            initpindir, PIN_INITIAL_DIR_OFFSET, PIN_BITFIELD_LENGTH) ;
    }
    /*
     * initpinvalue
     */
    Tcl_SetStringObj(keyObj, "initpinvalue", -1) ;
    if (Tcl_DictObjGet(interp, configObj, keyObj, &valueObj) != TCL_OK) {
        goto errorout ;
    }
    if (valueObj) {
        uint8 initpinvalue = EXTRACT_BITFIELD(newConfig.config.Pin,
            PIN_INITIAL_VALUE_OFFSET, PIN_BITFIELD_LENGTH) ;

        if (GetPinValueBitMask(interp, valueObj, &initpinvalue) != TCL_OK) {
            goto errorout ;
        }
        newConfig.config.Pin = INSERT_BITFIELD(newConfig.config.Pin,
            initpinvalue, PIN_INITIAL_VALUE_OFFSET, PIN_BITFIELD_LENGTH) ;
    }
    /*
     * closepindir
     */
    Tcl_SetStringObj(keyObj, "closepindir", -1) ;
```

```
    if (Tcl_DictObjGet(interp, configObj, keyObj, &valueObj) != TCL_OK) {
        goto errorout ;
    }
    if (valueObj) {
        uint8 closepindir = EXTRACT_BITFIELD(newConfig.config.Pin,
            PIN_CLOSE_DIR_OFFSET, PIN_BITFIELD_LENGTH) ;

        if (GetDirectionBitMask(interp, valueObj, &closepindir) != TCL_OK) {
            goto errorout ;
        }
        newConfig.config.Pin = INSERT_BITFIELD(newConfig.config.Pin,
            closepindir, PIN_CLOSE_DIR_OFFSET, PIN_BITFIELD_LENGTH) ;
    }
    /*
     * closepinvalue
     */
    Tcl_SetStringObj(keyObj, "closepinvalue", -1) ;
    if (Tcl_DictObjGet(interp, configObj, keyObj, &valueObj) != TCL_OK) {
        goto errorout ;
    }
    if (valueObj) {
        uint8 closepinvalue = EXTRACT_BITFIELD(newConfig.config.Pin,
            PIN_CLOSE_VALUE_OFFSET, PIN_BITFIELD_LENGTH) ;

        if (GetPinValueBitMask(interp, valueObj, &closepinvalue) != TCL_OK) {
            goto errorout ;
        }
        newConfig.config.Pin = INSERT_BITFIELD(newConfig.config.Pin,
            closepinvalue, PIN_CLOSE_VALUE_OFFSET, PIN_BITFIELD_LENGTH) ;
    }
    /*
     * transfer options
     */
    if (SetTransferOptions(interp, configObj, &newConfig.transferOptions)
            != TCL_OK) {        /* ❸ */
        goto errorout ;
    }

    *chanConfig = newConfig ;     /* ❹ */

    Tcl_DecrRefCount(keyObj) ;
    Tcl_ResetResult(interp) ;
    return TCL_OK ;

errorout:
    Tcl_DecrRefCount(keyObj) ;
    return TCL_ERROR ;
}

<<static data>>=
static char const setChannelConfigCmdName[] = "::mpssespi::setChannelConfig" ;
static char const setChannelConfigName[] = "setChannelConfig" ;

<<command creation>>=
Tcl_CreateObjCommand(interp, setChannelConfigCmdName, SetChannelConfigProc,
        clientData, NULL) ;
if (Tcl_Export(interp, ns, setChannelConfigName, 0) != TCL_OK) {
    goto errorout ;
}
if (Tcl_DictObjPut(interp, mapObj, Tcl_NewStringObj(setChannelConfigName, -1),
        Tcl_NewStringObj(setChannelConfigCmdName, -1)) != TCL_OK) {
    goto errorout ;
```

```
}
```

1. Here we take a copy of the channel configuration. As we decode the input dictionary argument, the values are placed in this copy.
2. Missing keys are just silently ignored.
3. Since transfer options may also be given on commands that cause SPI bus transactions, there is common code to deal with them.
4. Finally, we can overwrite the configuration with a modified value.

```
<<setChannelConfig tests>>=
test setChannelConfig-1.0 {
    set the channel configuration
} -setup {
    set handle [::mpssespi openChannel 0]
} -cleanup {
    ::mpssespi closeChannel $handle
} -body {
    ::mpssespi setChannelConfig $handle {xferunits bits}
    dict get [::mpssespi getChannelConfig $handle] xferunits
} -result {bits}
```

The code for this command performs some bit twiddling operations as it translates the configuration information from Tcl objects into the various bit fields and structure members of the `libMPSSE-SPI` channel configuration. Below are the macro definitions for all this twiddling. Note that we do this using “C” preprocessor macros. A more modern and arguably better approach would be to use `static inline` functions available in compilers supporting the C99 standard. However, in keeping with Tcl conventions, we tend to code to the lowest common denominator of compiler features.

```
<<macro definitions>>=
#define BITMASK(l)              ((1 << (l)) - 1)
#define FIELDMASK(o, l)         (BITMASK(l) << (o))
#define EXTRACT_BITFIELD(v, o, l)   (((v) & FIELDMASK(o, l)) >> (o))

#define CLEARFIELD(v, o, l)     ((v) & ~FIELDMASK(o, l))
#define ALIGNTOFIELD(f, o, l)   (((f) & BITMASK(l)) << (o))
#define INSERT_BITFIELD(v, f, o, l)\
    (CLEARFIELD(v, o, l) | ALIGNTOFIELD(f, o, l))

#define IS_BIT_XFER(o) (((o) & SPI_TRANSFER_OPTIONS_SIZE_IN_BITS) != 0)
```

Bit fields are defined by offset (`o`) and length (`l`). Offsets of 0 refer to the least significant bit of the operand. The length of a bit field is the number of contiguous bits that form the field. A bit mask is a value with 1’s in each bit position of the field and 0’s elsewhere.

We also need a few constants that are not provided in the library header files.

```
<<macro definitions>>=
#define PIN_BITFIELD_LENGTH         8

#define PIN_INITIAL_DIR_OFFSET      0
#define PIN_INITIAL_VALUE_OFFSET\
    (PIN_INITIAL_DIR_OFFSET + PIN_BITFIELD_LENGTH)
#define PIN_CLOSE_DIR_OFFSET\
    (PIN_INITIAL_VALUE_OFFSET + PIN_BITFIELD_LENGTH)
#define PIN_CLOSE_VALUE_OFFSET\
    (PIN_CLOSE_DIR_OFFSET + PIN_BITFIELD_LENGTH)
```

Note that the pin information is grouped together and we define the macros to emphasize that fact.

Pin Direction Configuration

Similar to what we did for the `getChannelConfig` command, we have factored out some common processing to deal with I/O pins.

The `GetDirectionBitMask()` function transforms a dictionary which specifies I/O pin direction into a bit mask that is suitable for use in the `libMPSSE-SPI` configuration.

The code uses a similar strategy as `GetDirectionDict()`

```
<<utility functions>>=
static int
GetDirectionBitMask(
    Tcl_Interp *interp,
    Tcl_Obj *dirSpecObj,
    uint8 *directionPtr)
{
    char key[2] ;
    Tcl_Obj *keyObj ;
    uint8 mask ;
    int count ;
    uint8 direction ;

    assert(directionPtr != NULL) ;
    direction = *directionPtr ;

    key[0] = '0' ;
    key[1] = '\0' ;
    keyObj = Tcl_NewStringObj(NULL, 0) ;

    for (mask = 1, count = 8 ; count > 0 ; mask <<= 1, count--) {
        Tcl_Obj *dirValueObj ;
        char const *dirValueStr ;

        Tcl_SetStringObj(keyObj, key, -1) ;
        key[0]++ ;

        if (Tcl_DictObjGet(interp, dirSpecObj, keyObj, &dirValueObj)
                != TCL_OK) {
            goto errorout ;
        }
        if (dirValueObj) {
            dirValueStr = Tcl_GetString(dirValueObj) ;
            if (strcmp(dirValueStr, "out") == 0) {
                direction |= mask ;
            } else if (strcmp(dirValueStr, "in") == 0) {
                direction &= ~mask ;
            } else {
                Tcl_SetObjResult(interp, Tcl_ObjPrintf(
                    "bad direction value, \"%s\": should be \"in\" or \"out\"",
                    dirValueStr)) ;
                goto errorout ;
            }
        }
    }
    Tcl_DecrRefCount(keyObj) ;

    *directionPtr =  direction ;

    return TCL_OK ;

errorout:
    Tcl_DecrRefCount(keyObj) ;
```

```
    return TCL_ERROR ;
}
```

We also need a function to handle the I/O pin values. This is the analog to `GetPinValueDict()`.

```
<<utility functions>>=
static int
GetPinValueBitMask(
    Tcl_Interp *interp,
    Tcl_Obj *lvalueSpecObj,
    uint8 *lvaluePtr)
{
    char key[2] ;
    Tcl_Obj *keyObj ;
    uint8 mask ;
    int count ;
    uint8 lvalue ;

    assert(lvaluePtr != NULL) ;
    lvalue = *lvaluePtr ;

    key[0] = '0' ;
    key[1] = '\0' ;
    keyObj = Tcl_NewStringObj(NULL, 0) ;

    for (mask = 1, count = 8 ; count > 0 ; mask <<= 1, count--) {
        Tcl_Obj *valueObj ;
        int pinValue ;

        Tcl_SetStringObj(keyObj, key, -1) ;
        key[0]++ ;

        if (Tcl_DictObjGet(interp, lvalueSpecObj, keyObj, &valueObj)
                != TCL_OK) {
            goto errorout ;
        }
        if (valueObj) {
            if (Tcl_GetBooleanFromObj(interp, valueObj, &pinValue) != TCL_OK) {
                goto errorout ;
            }
            if (pinValue) {
                lvalue |= mask ;
            } else {
                lvalue &= ~mask ;
            }
        }
    }
    Tcl_DecrRefCount(keyObj) ;

    *lvaluePtr =  lvalue ;

    return TCL_OK ;

errorout:
    Tcl_DecrRefCount(keyObj) ;
    return TCL_ERROR ;
}
```

Close Channel

When channels are no longer needed, they may be closed. Closing releases the resources acquired when the channel was opened.

`::mpssespi closeChannel` *handle*

handle
: A `mpssespi` channel handle as returned from a successful call to the `openChannel` command.

The `closeChannel` command closes the SPI channel associated with *handle*.

The code is a rather direct interface to `SPI_CloseChannel`.

```
<<command procedures>>=
static int
CloseChannelProc(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *const objv[])
{
    MPSSEPkgInfo *pkgInfo = (MPSSEPkgInfo *)clientData ;
    Tcl_Obj *handleObj ;
    MPSSEChanConfig *chanConfig ;
    FT_STATUS status ;
    int result ;

    if (objc == 2) {
        handleObj = objv[1] ;
    } else {
        Tcl_WrongNumArgs(interp, 1, objv, "channelHandle") ;
        return TCL_ERROR ;
    }

    if (LookUpHandle(interp, pkgInfo, handleObj, &chanConfig) != TCL_OK) {
        return TCL_ERROR ;
    }

#       ifdef TEST
    status = 0 ;
#       else
    status = SPI_CloseChannel(chanConfig->handle) ;
#       endif /* TEST */
    result = SetStatusResult(interp, status) ;

    if (result == TCL_OK) {
        DeleteHandleMapping(pkgInfo, handleObj) ; /* ❶ */
        Tcl_ResetResult(interp) ;
    }

    return result ;
}

<<static data>>=
static char const closeChannelCmdName[] = "::mpssespi::closeChannel" ;
static char const closeChannelName[] = "closeChannel" ;

<<command creation>>=
Tcl_CreateObjCommand(interp, closeChannelCmdName, CloseChannelProc,
        clientData, NULL) ;
if (Tcl_Export(interp, ns, closeChannelName, 0) != TCL_OK) {
    goto errorout ;
}
if (Tcl_DictObjPut(interp, mapObj, Tcl_NewStringObj(closeChannelName, -1),
        Tcl_NewStringObj(closeChannelCmdName, -1)) != TCL_OK) {
```

```
    goto errorout ;
}
```

(1) We clean up the handle mapping and configuration information here.

```
<<closeChannel tests>>=
test closeChannel-1.0 {
    close an open channel
} -setup {
    set handle [::mpssespi openChannel 0]
} -cleanup {
} -body {
    set handle [::mpssespi closeChannel $handle]
} -result {}
```

```
<<closeChannel tests>>=
test closeChannel-2.0 {
    close a non-existant channel
} -setup {
} -cleanup {
} -body {
    ::mpssespi closeChannel mpsse2000
} -result {unknown channel handle, "mpsse2000"} -returnCodes error
```

Read Channel

The `readChannel` command is provided to transfer data from a peripheral.

`::mpssespi readChannel` *handle size ?xferoptions?*

handle
A `mpssespi` channel handle as returned from a successful call to the `openChannel` command.

size
The number of bits or bytes to read. The units associated with the *size* value is determined by the transfer options configuration information.

xferoptions
A dictionary that specifies details of the read transaction. If the *xferoptions* argument is missing, then the previously specified values from the channel configuration are used. If present, the options values override the previous configuration for this bus transaction only. Not all keys need be supplied. Missing keys use the channel configuration. Unrecognized keys are silently ignored. The dictionary keys are the same as for that part of the channel configuration that deals with the transfer, namely:

`xferunits`
Specifies the units for which SPI bus data transfers are to happen. Valid values are "bits" or "bytes".

`csenable`
A boolean that specifies whether the chip select line is to be made active before the SPI bus transaction.

`csdisable`
A boolean that specifies whether the chip select line is to be made inactive after the SPI bus transaction.

The `readChannel` command performs a SPI bus transaction and returns a binary string containing the raw data read from the SPI bus. Note that the command is blocking until the specified amount of data is returned (or an error happens).

As you would expect, this command invokes `SPI_ReadChannel()` to perform the SPI bus transaction. To provide space for the data read in, we use a Tcl byte array.

```
<<command procedures>>=
static int
ReadChannelProc(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *const objv[])
{
    MPSSEPkgInfo *pkgInfo = (MPSSEPkgInfo *)clientData ;
    int xferSize ;
    MPSSEChanConfig *chanConfig ;
    uint32 transferOptions ;
    Tcl_Obj *inputObj ;
    unsigned char *inputBuf ;
    unsigned inBufByteSize ;
    uint32 xferActual ;
    FT_STATUS status ;
    int result ;

    if (objc < 3 || objc > 4) {
        Tcl_WrongNumArgs(interp, 1, objv, "channelHandle size ?xferoptions?") ;
        return TCL_ERROR ;
    }

    if (LookUpHandle(interp, pkgInfo, objv[1], &chanConfig) != TCL_OK) {
        return TCL_ERROR ;
    }

    if (chanConfig->isInitialized == 0) {
        SetInitializationError(interp) ;
        return TCL_ERROR ;
    }

    if (Tcl_GetIntFromObj(interp, objv[2], &xferSize) != TCL_OK) {
        return TCL_ERROR ;
    }

    /*
     * Start with the previously established configuration information
     * for the transfer options.
     */
    transferOptions = chanConfig->transferOptions ;
    if (objc == 4) {
        /*
         * If the additional argument is supplied, then we override
         * the options for this transaction only.
         */
        if (SetTransferOptions(interp, objv[3], &transferOptions) != TCL_OK) {
            return TCL_ERROR ;
        }
    }

    inputObj = Tcl_NewByteArrayObj(NULL, 0) ;    /* ❶ */
    inBufByteSize = ConvertXferUnitsToBytes(transferOptions, xferSize) ;
    inputBuf = Tcl_SetByteArrayLength(inputObj, inBufByteSize) ;

#       ifdef TEST
    status = 0 ;
    memset(inputBuf, 'A', inBufByteSize) ;
    xferActual = xferSize ;
#       else
```

```
    status = SPI_Read(chanConfig->handle, inputBuf, xferSize, &xferActual,
            transferOptions) ;
    result = SetStatusResult(interp, status) ;
#       endif /* TEST */

    if (result == TCL_OK) {
        if (IS_BIT_XFER(transferOptions)) {
            int residueBits = xferActual % 8 ;
            if (residueBits != 0) {           /* 2 */
                inputBuf[xferActual / 8] <<= 8 - residueBits ;
            }
        }
        Tcl_SetByteArrayLength(inputObj,
                ConvertXferUnitsToBytes(transferOptions, xferActual)) ;/* 3 */
        Tcl_SetObjResult(interp, inputObj) ;
    } else {
        Tcl_DecrRefCount(inputObj) ;
    }

    return result ;
}

<<static data>>=
static char const readChannelCmdName[] = "::mpssespi::readChannel" ;
static char const readChannelName[] = "readChannel" ;

<<command creation>>=
Tcl_CreateObjCommand(interp, readChannelCmdName, ReadChannelProc,
        clientData, NULL) ;
if (Tcl_Export(interp, ns, readChannelName, 0) != TCL_OK) {
    goto errorout ;
}
if (Tcl_DictObjPut(interp, mapObj, Tcl_NewStringObj(readChannelName, -1),
        Tcl_NewStringObj(readChannelCmdName, -1)) != TCL_OK) {
    goto errorout ;
}
```

1. The returned result will be a byte array. The length is determined by how much data was requested.
2. For bit transfers, if the number of bits transferred is not a byte multiple, then we shift the residual bits to the upper part of the byte. This recognizes that the order is most significant bits first and makes the resulting bit stream continguous in the byte array.
3. We have to set the length of the returned byte array to account for what was actually transferred.

```
<<readChannel tests>>=
test readChannel-1.0 {
    read a channel
} -setup {
    set handle [::mpssespi openChannel 0]
} -cleanup {
    ::mpssespi closeChannel $handle
} -body {
    set read [::mpssespi readChannel $handle 10]
    string index $read 0
} -result {A}
```

```
<<readChannel tests>>=
test readChannel-2.0 {
    read a channel with transfer options
```

```
} -setup {
    set handle [::mpssespi openChannel 0]
} -cleanup {
    ::mpssespi closeChannel $handle
} -body {
    set read [::mpssespi readChannel $handle 10 {csenable false}]
    string index $read 0
} -result {A}
```

The `SetTransferOptions` function takes a set of transfer options and overlays them with any user supplied options. User supplied options take precedence. This function is used in a number of places where `libMPSSE-SPI` allows specifying transfer options, *i.e.* on all the I/O functions.

The code itself deals with the dictionary of transfer options in much the same pattern as we have already seen.

```
<<utility functions>>=
static int
SetTransferOptions(
    Tcl_Interp *interp,
    Tcl_Obj *optionsDict,
    uint32 *optionsPtr)
{
    uint32 transferOptions ;
    Tcl_Obj *key ;
    Tcl_Obj *optValue ;
    char const *unitsString ;
    int csopt ;

    assert(optionsDict != NULL) ;
    assert(optionsPtr != NULL) ;

    transferOptions = *optionsPtr ;

    key = Tcl_NewStringObj("xferunits", -1) ;
    if (Tcl_DictObjGet(interp, optionsDict, key, &optValue) != TCL_OK) {
        goto errorout ;
    }
    if (optValue) {
        unitsString = Tcl_GetString(optValue) ;
        if (strcmp(unitsString, "bits") == 0) {
            transferOptions &= ~SPI_TRANSFER_OPTIONS_SIZE_IN_BITS ;
            transferOptions |= SPI_TRANSFER_OPTIONS_SIZE_IN_BITS ;
        } else if (strcmp(unitsString, "bytes") == 0) {
            transferOptions &= ~SPI_TRANSFER_OPTIONS_SIZE_IN_BYTES ;
            transferOptions |= SPI_TRANSFER_OPTIONS_SIZE_IN_BYTES ;
        } else {
            Tcl_SetObjResult(interp, Tcl_ObjPrintf(
                "unknown \"xferunits\" option value, \"%s\": should be \"bits\" or \"bytes ↩
                    \"", unitsString)) ;
            goto errorout ;
        }
    }
    Tcl_DecrRefCount(key) ;

    key = Tcl_NewStringObj("csenable", -1) ;
    if (Tcl_DictObjGet(interp, optionsDict, key, &optValue) != TCL_OK) {
        goto errorout ;
    }
    if (optValue) {
        if (Tcl_GetBooleanFromObj(interp, optValue, &csopt) != TCL_OK) {
            goto errorout ;
        }
```

```
        if (csopt) {
            transferOptions |= SPI_TRANSFER_OPTIONS_CHIPSELECT_ENABLE ;
        } else {
            transferOptions &= ~SPI_TRANSFER_OPTIONS_CHIPSELECT_ENABLE ;
        }
    }
    Tcl_DecrRefCount(key) ;

    key = Tcl_NewStringObj("csdisable", -1) ;
    if (Tcl_DictObjGet(interp, optionsDict, key, &optValue) != TCL_OK) {
        goto errorout ;
    }
    if (optValue) {
        if (Tcl_GetBooleanFromObj(interp, optValue, &csopt) != TCL_OK) {
            goto errorout ;
        }
        if (csopt) {
            transferOptions |= SPI_TRANSFER_OPTIONS_CHIPSELECT_DISABLE ;
        } else {
            transferOptions &= ~SPI_TRANSFER_OPTIONS_CHIPSELECT_DISABLE ;
        }
    }
    Tcl_DecrRefCount(key) ;

    *optionsPtr = transferOptions ;
    return TCL_OK ;

errorout:
    Tcl_DecrRefCount(key) ;
    return TCL_ERROR ;
}
```

We also factor out here a small function to round up a number of bits to the nearest whole byte. The check against the transfer options insures that the conversion is necessary.

```
<<utility functions>>=
static int
ConvertXferUnitsToBytes(
    uint32 transferOptions,
    uint32 xferSize)
{
    return IS_BIT_XFER(transferOptions) ? (xferSize + 7) / 8 : xferSize ;
}
```

Write Channel

The `writeChannel` command is provided to transfer data to a peripheral.

`::mpssespi writeChannel` *handle outdata ?xferoptions? ?nbits?*

handle
A `mpssespi` channel handle as returned from a successful call to the `openChannel` command.

outdata
The data to be written to the SPI bus. This is interpreted as a byte array and the raw byte values contained in this argument are written directly to the SPI bus.

xferoptions
A dictionary that specifies details of the read transaction. If the *xferoptions* argument is missing, then the previously specified values from the channel configuration are used. If present, the options values override the previous configuration only for this transaction. Not all keys need be supplied. Missing keys use the previously established channel configuration. Extraneous keys are *silently ignored*. The dictionary keys are the same as for that part of the channel configuration that deals with the transfer, namely:

`xferunits`
Specifies the units for which SPI bus data transfers are to happen. Valid values are "bits" or "bytes".

`csenable`
A boolean that specifies whether the chip select line is to be made active before the SPI bus transaction.

`csdisable`
A boolean that specifies whether the chip select line is to be made inactive after the SPI bus transaction.

nbits
For SPI bus writes where the length of the transfer is specified in bits (*i.e.* the `xferunits` transfer option has the value `bits`), This argument specifies the number of bits in *outdata* to transfer. If the argument is missing, then all the bytes contained in *outdata* are transferred (*i.e.* the number of bits transferred is the length of *outdata* in bytes times eight). Otherwise, only the first *nbits* of *outdata* is written to the bus. If transfer units are specified as `bytes`, then the value of this argument is ignored.

The `writeChannel` command performs a SPI bus transaction clocking *outdata* to the bus. The return value of the command is an integer value giving the amount of data actually transferred. The units of that data will be the same as that specified by the `xferunits` transfer option configuration. Note that the command is blocking until the specified amount of data is transferred (or an error happens).

Following the same pattern as the `readChannel` command, this code is primarily an interface to the `SPI_WriteChannel()` function.

```
<<command procedures>>=
static int
WriteChannelProc(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *const objv[])
{
    MPSSEPkgInfo *pkgInfo = (MPSSEPkgInfo *)clientData ;
    MPSSEChanConfig *chanConfig ;
    uint32 transferOptions ;
    unsigned char *outputBuf ;
    int outBufLen ;
    int bitLen ;
    int xferSize ;
    uint32 xferActual ;
    FT_STATUS status ;
    int result ;

    if (objc < 3 || objc > 5) {
        Tcl_WrongNumArgs(interp, 1, objv,
```

```
                "channelHandle outdata ?xferoptions? ?nbits?") ;
        return TCL_ERROR ;
    }

    if (LookUpHandle(interp, pkgInfo, objv[1], &chanConfig) != TCL_OK) {
        return TCL_ERROR ;
    }

    if (chanConfig->isInitialized == 0) {
        SetInitializationError(interp) ;
        return TCL_ERROR ;
    }

    outputBuf = Tcl_GetByteArrayFromObj(objv[2], &outBufLen) ;

    transferOptions = chanConfig->transferOptions ;
    if (objc >= 4) {
        if (SetTransferOptions(interp, objv[3], &transferOptions) != TCL_OK) {
            return TCL_ERROR ;
        }
    }

    if (objc == 5) {
        if (Tcl_GetIntFromObj(interp, objv[4], &bitLen) != TCL_OK) {
            return TCL_ERROR ;
        }
    } else {
        bitLen = outBufLen * 8 ;
    }
    if (VerifyOutputLength(interp, transferOptions, outBufLen, bitLen) !=
            TCL_OK) {
        return TCL_ERROR ;
    }
    xferSize = IS_BIT_XFER(transferOptions) ? bitLen : outBufLen ;

#       ifdef TEST
    (void)outputBuf ;
    status = 0 ;
    xferActual = xferSize ;
#       else
    status = SPI_Write(chanConfig->handle, outputBuf, xferSize, &xferActual,
            transferOptions) ;
    result = SetStatusResult(interp, status) ;
#       endif /* TEST */

    result = SetStatusResult(interp, status) ;
    if (result == TCL_OK) {
        Tcl_SetObjResult(interp, Tcl_NewIntObj(xferActual)) ;
    }

    return result ;
}

<<static data>>=
static char const writeChannelCmdName[] = "::mpssespi::writeChannel" ;
static char const writeChannelName[] = "writeChannel" ;

<<command creation>>=
Tcl_CreateObjCommand(interp, writeChannelCmdName, WriteChannelProc,
        clientData, NULL) ;
if (Tcl_Export(interp, ns, writeChannelName, 0) != TCL_OK) {
    goto errorout ;
```

```
}
if (Tcl_DictObjPut(interp, mapObj, Tcl_NewStringObj(writeChannelName, -1),
        Tcl_NewStringObj(writeChannelCmdName, -1)) != TCL_OK) {
    goto errorout ;
}
```

One minor complication in writing is we need to make sure that if the user has specified that the units of transfer is to be *bits* that he has given us a buffer that contains at least that many bits. For transfers in *bytes* units, we can simply take the lenght of the tranfer. Here we have factored out the verification that the user input is correct. We will use this function again later.

```
<<utility functions>>=
static int
VerifyOutputLength(
    Tcl_Interp *interp,
    uint32 transferOptions,
    int byteLen,
    int bitLen)
{
    /*
     * We need to verify that if the transfer is specified in bit units that
     * the actual length of the byte array is long enough to contain that
     * number of bytes.
     */
    if (IS_BIT_XFER(transferOptions) && (byteLen < (bitLen + 7) / 8)) {
        Tcl_SetObjResult(interp, Tcl_ObjPrintf(
            "output data is %d bytes long, \
however %d bits were requested to be written",
            byteLen, bitLen)) ;
        return TCL_ERROR ;
    }
    return TCL_OK ;
}
```

```
<<writeChannel tests>>=
test writeChannel-1.0 {
    write a channel
} -setup {
    set handle [::mpssespi openChannel 0]
} -cleanup {
    ::mpssespi closeChannel $handle
} -body {
    set outdata CCCCC
    set written [::mpssespi writeChannel $handle $outdata]
    expr {$written == [string length $outdata]}
} -result {1}
```

```
<<writeChannel tests>>=
test writeChannel-2.0 {
    write a channel using bits
} -setup {
    set handle [::mpssespi openChannel 0]
} -cleanup {
    ::mpssespi closeChannel $handle
} -body {
    set outdata CCCCC
    ::mpssespi writeChannel $handle $outdata {xferunits bits} 10
} -result {10}
```

```
<<writeChannel tests>>=
test writeChannel-3.0 {
    write a channel using bits but an invalid length
```

```
} -setup {
    set handle [::mpssespi openChannel 0]
} -cleanup {
    ::mpssespi closeChannel $handle
} -body {
    set outdata CCCCC
    ::mpssespi writeChannel $handle $outdata {xferunits bits} 1000
} -result {output data is 5 bytes long, however 1000 bits were requested to be written} ↩
    -returnCodes error
```

Read/Write Channel

A SPI bus clocks data out of the master into a slave (peripheral chip) and out of a slave into the master simultaneously. Many SPI bus peripherals use this feature and so a command that will both read and write is required. The `readWriteChannel` command both transfers data out and returns the data clocked out of the slave.

`::mpssespi readWriteChannel` *handle outdata ?xferoptions? ?nbits?*

handle
A `mpssespi` channel handle as returned from a successful call to the `openChannel` command.

outdata
The data to be written to the SPI bus. This is interpreted as a byte array and the raw byte values contained in this argument are written directly to the SPI bus.

xferoptions
A dictionary that specifies details of the read transaction. If the *xferoptions* argument is missing, then the previously specified values from the channel configuration are used. If present, the options values override the previous configuration only for this transaction. Not all keys need be supplied. Missing keys use the previously established channel configuration. Extraneous keys are *silently ignored*. The dictionary keys are the same as for that part of the channel configuration that deals with the transfer, namely:

`xferunits`
Specifies the units for which SPI bus data transfers are to happen. Valid values are "bits" or "bytes".

`csenable`
A boolean that specifies whether the chip select line is to be made active before the SPI bus transaction.

`csdisable`
A boolean that specifies whether the chip select line is to be made inactive after the SPI bus transaction.

nbits
For SPI bus writes where the length of the transfer is specified in bits (*i.e.* the `xferunits` transfer option has the value `bits`), This argument specifies the number of bits in *outdata* to transfer. If the argument is missing, then all the bytes contained in *outdata* are transferred (*i.e.* the number of bits transferred is the length of *outdata* in bytes times eight). Otherwise, only the first *nbits* of *outdata* is written to the bus. If transfer units are specified as `bytes`, then the value of this argument is ignored. *N.B.* see the known problem with this command.

The `readWriteChannel` command performs a SPI bus transaction clocking *outdata* to the bus. The return value of the command is a byte array value giving the data returned by the slave during the bus transaction. Note that the command is blocking until the specified amount of data is transferred (or an error happens).

This code follows the pattern established above.

```
<<command procedures>>=
static int
ReadWriteChannelProc(
    ClientData clientData,
    Tcl_Interp *interp,
```

```
    int objc,
    Tcl_Obj *const objv[])
{
    MPSSEPkgInfo *pkgInfo = (MPSSEPkgInfo *)clientData ;
    MPSSEChanConfig *chanConfig ;
    uint32 transferOptions ;
    Tcl_Obj *inputObj ;
    unsigned char *inputBuf ;
    unsigned char *outputBuf ;
    int outBufLen ;
    int bitLen ;
    int xferSize ;
    uint32 xferActual ;
    FT_STATUS status ;
    int result ;

    if (objc < 3 || objc > 5) {
        Tcl_WrongNumArgs(interp, 1, objv,
                "channelHandle outdata ?xferoptions? ?nbits?") ;
        return TCL_ERROR ;
    }

    if (LookUpHandle(interp, pkgInfo, objv[1], &chanConfig) != TCL_OK) {
        return TCL_ERROR ;
    }

    if (chanConfig->isInitialized == 0) {
        SetInitializationError(interp) ;
        return TCL_ERROR ;
    }

    outputBuf = Tcl_GetByteArrayFromObj(objv[2], &outBufLen) ;

    transferOptions = chanConfig->transferOptions ;
    if (objc >= 4) {
        if (SetTransferOptions(interp, objv[3], &transferOptions) != TCL_OK) {
            return TCL_ERROR ;
        }
    }

    if (objc == 5) {
        if (Tcl_GetIntFromObj(interp, objv[4], &bitLen) != TCL_OK) {
            return TCL_ERROR ;
        }
    } else {
        bitLen = outBufLen * 8 ;
    }
    if (VerifyOutputLength(interp, transferOptions, outBufLen, bitLen) !=
            TCL_OK) {
        return TCL_ERROR ;
    }
    xferSize = IS_BIT_XFER(transferOptions) ? bitLen : outBufLen ;
    inputObj = Tcl_NewByteArrayObj(NULL, 0) ;
    inputBuf = Tcl_SetByteArrayLength(inputObj, outBufLen) ;

#       ifdef TEST
    (void)outputBuf ;
    (void)inputBuf ;
    status = 0 ;
    memset(inputBuf, 'B', outBufLen) ;
    xferActual = xferSize ;
#       else
```

```
    status = SPI_ReadWrite(chanConfig->handle, inputBuf, outputBuf,
            xferSize, &xferActual, transferOptions) ;
    result = SetStatusResult(interp, status) ;
#       endif /* TEST */

    result = SetStatusResult(interp, status) ;
    if (result == TCL_OK) {
        if (IS_BIT_XFER(transferOptions)) {
            int residueBits = xferActual % 8 ;
            if (residueBits != 0) {
                inputBuf[xferActual / 8] <<= 8 - residueBits ;
            }
        }
        Tcl_SetByteArrayLength(inputObj,
                ConvertXferUnitsToBytes(transferOptions, xferActual)) ;
        Tcl_SetObjResult(interp, inputObj) ;
    } else {
        Tcl_DecrRefCount(inputObj) ;
    }

    return result ;
}

<<static data>>=
static char const readWriteChannelCmdName[] = "::mpssespi::readWriteChannel" ;
static char const readWriteChannelName[] = "readWriteChannel" ;

<<command creation>>=
Tcl_CreateObjCommand(interp, readWriteChannelCmdName, ReadWriteChannelProc,
        clientData, NULL) ;
if (Tcl_Export(interp, ns, readWriteChannelName, 0) != TCL_OK) {
    goto errorout ;
}
if (Tcl_DictObjPut(interp, mapObj, Tcl_NewStringObj(readWriteChannelName, -1),
        Tcl_NewStringObj(readWriteChannelCmdName, -1)) != TCL_OK) {
    goto errorout ;
}
```

```
<<readWriteChannel tests>>=
test readWriteChannel-1.0 {
    read/write a channel
} -setup {
    set handle [::mpssespi openChannel 0]
} -cleanup {
    ::mpssespi closeChannel $handle
} -body {
    set outdata CCCCC
    set read [::mpssespi readWriteChannel $handle $outdata]
    string index $read 0
} -result {B}
```

Is Busy

This command is a direct interface to the `SPI_IsBusy()` function. It is used to poll the state of the MISO line without clocking the bus. Some SPI peripherals use the state of the MISO to indicate status or readyness.

`::mpssespi isBusy` *handle*

handle
A `mpssespi` channel handle as returned from a successful call to the `openChannel` command.

The `isBusy` command polls the state of the MISO line without clocking the bus. The return value is a boolean indicating whether the channel was busy.

The code for this command is a direct interface to the `SPI_IsBusy()` function.

```
<<command procedures>>=
static int
IsBusyProc(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *const objv[])
{
    MPSSEPkgInfo *pkgInfo = (MPSSEPkgInfo *)clientData ;
    MPSSEChanConfig *chanConfig ;
    bool busyStatus ;
    FT_STATUS status ;
    int result ;

    if (objc != 2) {
        Tcl_WrongNumArgs(interp, 1, objv, "channelHandle") ;
        return TCL_ERROR ;
    }

    if (LookUpHandle(interp, pkgInfo, objv[1], &chanConfig) != TCL_OK) {
        return TCL_ERROR ;
    }

    if (chanConfig->isInitialized == 0) {
        SetInitializationError(interp) ;
        return TCL_ERROR ;
    }

#       ifdef TEST
    status = 0 ;
    busyStatus = 0 ;
#       else
    status = SPI_IsBusy(chanConfig->handle, &busyStatus) ;
#       endif /* TEST */
    result = SetStatusResult(interp, status) ;
    if (result == TCL_OK) {
        Tcl_SetObjResult(interp, Tcl_NewBooleanObj(busyStatus)) ;
    }

    return result ;
}

<<static data>>=
static char const isBusyCmdName[] = "::mpssespi::isBusy" ;
static char const isBusyName[] = "isBusy" ;

<<command creation>>=
Tcl_CreateObjCommand(interp, isBusyCmdName, IsBusyProc,
        clientData, NULL) ;
if (Tcl_Export(interp, ns, isBusyName, 0) != TCL_OK) {
```

```
    goto errorout ;
}
if (Tcl_DictObjPut(interp, mapObj, Tcl_NewStringObj(isBusyName, -1),
        Tcl_NewStringObj(isBusyCmdName, -1)) != TCL_OK) {
    goto errorout ;
}
```

```
<<isBusy tests>>=
test isBusy-1.0 {
    check if a channel is busy
} -setup {
    set handle [::mpssespi openChannel 0]
} -cleanup {
    ::mpssespi closeChannel $handle
} -body {
    ::mpssespi isBusy $handle
} -result {0}
```

Change CS

The `libMPSSE-SPI` library provide a second way to change some of the channel configuration. In addition to the `initChannel` command, the `changeCS` command can be used to modify the channel configuration. However, only that part of the configuration pertaining to the chip select and SPI bus mode is changed. So, invoking the `setChannelConfig` command to change the configuration of **mode**, **csline** or **csactive** followed by invoking the `changeCS` command will install new values for that portion of the channel configuration.

`::mpssespi changeCS` *handle*

handle
: A `mpssespi` channel handle as returned from a successful call to the `openChannel` command.

The `changeCS` command changes **mode**, **csline** and **csactive** configuration of the channel given by *handle* without having to reinitialize the channel. For hardware configurations where multiple SPI peripherals are connected to the same FTDI chip, this command can be used to switch the active chip select line. The return value is the empty string.

Again the code is just a direct interface to the `SPI_ChangeCS()` function.

```
<<command procedures>>=
static int
ChangeCSProc(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *const objv[])
{
    MPSSEPkgInfo *pkgInfo = (MPSSEPkgInfo *)clientData ;
    MPSSEChanConfig *chanConfig ;
    FT_STATUS status ;
    int result ;

    if (objc != 2) {
        Tcl_WrongNumArgs(interp, 1, objv, "channelHandle") ;
        return TCL_ERROR ;
    }

    if (LookUpHandle(interp, pkgInfo, objv[1], &chanConfig) != TCL_OK) {
        return TCL_ERROR ;
    }
```

```
    if (chanConfig->isInitialized == 0) {
        SetInitializationError(interp) ;
        return TCL_ERROR ;
    }

#       ifdef TEST
    status = 0 ;
#       else
    status = SPI_ChangeCS(chanConfig->handle, chanConfig->config.configOptions) ;
#       endif /* TEST */
    result = SetStatusResult(interp, status) ;
    if (result == TCL_OK) {
        Tcl_ResetResult(interp) ;
    }

    return result ;
}

<<static data>>=
static char const changeCSCmdName[] = "::mpssespi::changeCS" ;
static char const changeCSName[] = "changeCS" ;

<<command creation>>=
Tcl_CreateObjCommand(interp, changeCSCmdName, ChangeCSProc,
        clientData, NULL) ;
if (Tcl_Export(interp, ns, changeCSName, 0) != TCL_OK) {
    goto errorout ;
}
if (Tcl_DictObjPut(interp, mapObj, Tcl_NewStringObj(changeCSName, -1),
        Tcl_NewStringObj(changeCSCmdName, -1)) != TCL_OK) {
    goto errorout ;
}
```

```
<<changeCS tests>>=
test changeCS-1.0 {
    change config options
} -setup {
    set handle [::mpssespi openChannel 0]
} -cleanup {
    ::mpssespi closeChannel $handle
} -body {
    ::mpssespi changeCS $handle
} -result {}
```

Write GPIO

The FTDI chips also have I/O pins associated with them that can be controlled via the SPI library.

`::mpssespi writeGPIO` *handle direction values*

handle
A `mpssespi` channel handle as returned from a successful call to the `openChannel` command.

direction
A dictionary with keys "0", "1" … "7" corresponding to the 8 GPIO lines and with values of either "in" or "out". Any missing GPIO line key is set to "in".

values
A dictionary with keys "0", "1" .. "7" and values that are boolean values corresponding to logic low (false) and logic high (true). Any missing GPIO line value is set to "false".

The `writeGPIO` command controls the direction and state of the eight GPIO lines associated with the SPI channel. The return value of this command is the empty string.

The command is another direct interface to a library function.

```
<<command procedures>>=
static int
WriteGPIOProc(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *const objv[])
{
    MPSSEPkgInfo *pkgInfo = (MPSSEPkgInfo *)clientData ;
    MPSSEChanConfig *chanConfig ;
    uint8 dir ;
    uint8 lvalue ;
    FT_STATUS status ;
    int result ;

    if (objc != 4) {
        Tcl_WrongNumArgs(interp, 1, objv, "channelHandle direction value") ;
        return TCL_ERROR ;
    }

    if (LookUpHandle(interp, pkgInfo, objv[1], &chanConfig) != TCL_OK) {
        return TCL_ERROR ;
    }

    if (chanConfig->isInitialized == 0) {
        SetInitializationError(interp) ;
        return TCL_ERROR ;
    }

    dir = 0 ;
    if (GetDirectionBitMask(interp, objv[2], &dir) != TCL_OK) { /* ❶ */
        return TCL_ERROR ;
    }

    lvalue = 0 ;
    if (GetPinValueBitMask(interp, objv[3], &lvalue) != TCL_OK) {
        return TCL_ERROR ;
    }

#       ifdef TEST
    status = 0 ;
#       else
```

```
    status = FT_WriteGPIO(chanConfig->handle, dir, lvalue) ;
#       endif /* TEST */
    result = SetStatusResult(interp, status) ;
    if (result == TCL_OK) {
        Tcl_ResetResult(interp) ;
    }

    return result ;
}

<<static data>>=
static char const writeGPIOCmdName[] = "::mpssespi::writeGPIO" ;
static char const writeGPIOName[] = "writeGPIO" ;

<<command creation>>=
Tcl_CreateObjCommand(interp, writeGPIOCmdName, WriteGPIOProc,
        clientData, NULL) ;
if (Tcl_Export(interp, ns, writeGPIOName, 0) != TCL_OK) {
    goto errorout ;
}
if (Tcl_DictObjPut(interp, mapObj, Tcl_NewStringObj(writeGPIOName, -1),
        Tcl_NewStringObj(writeGPIOCmdName, -1)) != TCL_OK) {
    goto errorout ;
}
```

1. The function for handling I/O pin directions and values were also used for setting configuration information.

```
<<writeGPIO tests>>=
test writeGPIO-1.0 {
    output GPIO values
} -setup {
    set handle [::mpssespi openChannel 0]
} -cleanup {
    ::mpssespi closeChannel $handle
} -body {
    ::mpssespi writeGPIO $handle {0 out 2 out} {0 true 2 false}
} -result {}
```

Read GPIO

The is a corresponding function to read the value of the I/O pins

`::mpssespi readGPIO` *handle*

handle
: A `mpssespi` channel handle as returned from a successful call to the `openChannel` command.

The `readGPIO` command reads and state of the eight GPIO lines associated with the SPI channel. The return value of this command is a dictionary with keys "0", "1" ... "7" corresponding to the 8 GPIO lines and with values that are boolean corresponding to logic low (false) and logic high (true).

```
<<command procedures>>=
static int
ReadGPIOProc(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,
```

```
    Tcl_Obj *const objv[])
{
    MPSSEPkgInfo *pkgInfo = (MPSSEPkgInfo *)clientData ;
    MPSSEChanConfig *chanConfig ;
    uint8 gpiovalue ;
    Tcl_Obj *valueDict ;
    FT_STATUS status ;
    int result ;

    if (objc != 2) {
        Tcl_WrongNumArgs(interp, 1, objv, "channelHandle") ;
        return TCL_ERROR ;
    }

    if (LookUpHandle(interp, pkgInfo, objv[1], &chanConfig) != TCL_OK) {
        return TCL_ERROR ;
    }

    if (chanConfig->isInitialized == 0) {
        SetInitializationError(interp) ;
        return TCL_ERROR ;
    }

#       ifdef TEST
    status = 0 ;
    gpiovalue = 0xa5 ;
#       else
    status = FT_ReadGPIO(chanConfig->handle, &gpiovalue) ;
#       endif /* TEST */
    result = SetStatusResult(interp, status) ;
    if (result != TCL_OK) {
        return TCL_ERROR ;
    }

    if (GetPinValueDict(interp, gpiovalue, &valueDict) != TCL_OK) {
        return TCL_ERROR ;
    }

    Tcl_SetObjResult(interp, valueDict) ;
    return TCL_OK ;
}

<<static data>>=
static char const readGPIOCmdName[] = "::mpssespi::readGPIO" ;
static char const readGPIOName[] = "readGPIO" ;

<<command creation>>=
Tcl_CreateObjCommand(interp, readGPIOCmdName, ReadGPIOProc,
        clientData, NULL) ;
if (Tcl_Export(interp, ns, readGPIOName, 0) != TCL_OK) {
    goto errorout ;
}
if (Tcl_DictObjPut(interp, mapObj, Tcl_NewStringObj(readGPIOName, -1),
        Tcl_NewStringObj(readGPIOCmdName, -1)) != TCL_OK) {
    goto errorout ;
}
```

```
<<readGPIO tests>>=
test readGPIO-1.0 {
    input GPIO values
} -setup {
    set handle [::mpssespi openChannel 0]
```

```
} -cleanup {
    ::mpssespi closeChannel $handle
} -body {
    set gpios [::mpssespi readGPIO $handle]
    dict get $gpios 7
} -result {1}
```

Error Handling

All the handling of errors returned directly from `libMPSSE-SPI` is factored into the `SetStatusResult()` function.

```
<<utility functions>>=
static int
SetStatusResult(
    Tcl_Interp *interp,
    FT_STATUS status)
{
    static struct ftErrorMap {
        char const *resultString ;
        char const *errorCodeString ;
    } errorMap[] = {
        {"Ok", "FT_OK"},            /* place holder */
        {"invalid handle", "FT_INVALID_HANDLE"},
        {"device not found", "FT_DEVICE_NOT_FOUND"},
        {"device not opened", "FT_DEVICE_NOT_OPENED"},
        {"I/O error", "FT_IO_ERROR"},
        {"insufficient resources", "FT_INSUFFICIENT_RESOURCES"},
        {"invalid parameter", "FT_INVALID_PARAMETER"},
        {"invalid baud rate", "FT_INVALID_BAUD_RATE"},
        {"device not opened for erase", "FT_DEVICE_NOT_OPENED_FOR_ERASE"},
        {"device not opened for writing", "FT_DEVICE_NOT_OPENED_FOR_WRITE"},
        {"failed to write to device", "FT_FAILED_TO_WRITE_DEVICE"},
        {"failed to read from device", "FT_EEPROM_READ_FAILED"},
        {"EEPROM write failed", "FT_EEPROM_WRITE_FAILED"},
        {"EEPROM erase failed", "FT_EEPROM_ERASE_FAILED"},
        {"EEPROM not present", "FT_EEPROM_NOT_PRESENT"},
        {"EEPROM not programmed", "FT_EEPROM_NOT_PROGRAMMED"},
        {"invalid arguments", "FT_INVALID_ARGS"},
        {"not supported", "FT_NOT_SUPPORTED"},
        {"other error", "FT_OTHER_ERROR"},
    } ;

    if (FT_SUCCESS(status)) {
        return TCL_OK ;
    }

    Tcl_ResetResult(interp) ;

    assert(status < COUNTOF(errorMap)) ;

    if (status < COUNTOF(errorMap)) {
        Tcl_AppendResult(interp, errorMap[status].resultString, NULL) ;
        Tcl_SetErrorCode(interp, "MPSSESPI", errorMap[status].errorCodeString,
                errorMap[status].resultString, NULL) ; /* ❶ */
    } else {
        Tcl_Obj *result = Tcl_GetObjResult(interp) ;
        Tcl_AppendPrintfToObj(result, "unknown FT_STATUS value, \"%d\"",
            status) ;
        Tcl_SetObjResult(interp, result) ;
```

```
    }
    return TCL_ERROR ;
}
```

❶ We add error code information that precisely defines the error and is programatically convenient.

Since we track the state of the channel initialization, we can emit an error message for those operations that require initialization first.

```
<<utility functions>>=
static void
SetInitializationError(
    Tcl_Interp *interp)
{
    static char const msg[] = "channel must be initialized first" ;
    Tcl_SetObjResult(interp, Tcl_NewStringObj(msg, -1)) ;
    Tcl_SetErrorCode(interp, "MPSSESPI", "UNINITIALIZED", msg, NULL) ;
}
```

Package Initialization

Dynamically loaded packages under Tcl follow a naming convention for the initialization functions. This enables the `load` command to construct the initialization invocation from the package name.

Initialization is also separated for *safe* interpreters. The sections below show that initialization.

Load Initialization

```
<<initialization>>=
DLLEXPORT       /* ❶ */
int
Mpssespi_Init(
    Tcl_Interp *interp)
{
    ClientData clientData ;
    Tcl_Namespace *ns ;
    Tcl_Obj *mapObj ;
    Tcl_Command cmdToken ;

    if (Tcl_InitStubs(interp, TCL_VERSION, 0) == NULL) {
        return TCL_ERROR ;
    }

    clientData = NewMPSSEPkgInfo(interp) ;  /* ❷ */

    Init_libMPSSE() ;

    <<namespace creation>>
    <<command creation>>
    <<ensemble creation>>

    <<package configuration>>

    Tcl_PkgProvide(interp, PACKAGE_NAME, PACKAGE_VERSION) ;
    return TCL_OK ;
```

```
errorout:
    Tcl_DecrRefCount(mapObj) ;
    Tcl_DeleteNamespace(ns) ;
    return TCL_ERROR ;
}
```

1. Needed to build under Windows.
2. We create new package specific storage and pass it's pointer as the `clientData` to each command procedure.

Creating the Package Namespace

The `mpssespi` package is an ensemble command with subcommands corresponding to the interfaces to `libMPSSE-SPI`. The ensemble command is bound to the `::mpssespi` namespace.

```
<<static data>>=
static char const mpssespi_ns_name[] = "::mpssespi" ;

<<namespace creation>>=
ns = Tcl_CreateNamespace(interp, mpssespi_ns_name, NULL, NULL) ;
mapObj = Tcl_NewDictObj() ;      /* 1 */
```

1. We also obtain a dictionary object that is used to create the ensemble command mapping.

Once all the commands have been created and exported, we can create the ensemble command and install the command map.

```
<<ensemble creation>>=
cmdToken = Tcl_CreateEnsemble(interp, mpssespi_ns_name, ns, TCL_ENSEMBLE_PREFIX) ;
if (Tcl_SetEnsembleMappingDict(interp, cmdToken, mapObj) != TCL_OK) {
    goto errorout ;
}
```

Unloading

Unloading is supported and we use this as a good place to call the library clean up function.

```
<<unloading>>=
DLLEXPORT int
Mpssespi_Unload(
    Tcl_Interp *interp)
{
    Cleanup_libMPSSE() ;
    return TCL_OK ;
}
```

Safe Interpreter Initialization

Loading this package into a safe interpreter is *not* supported. The initialization and unloading functions just return an error.

```
<<initialization>>=
DLLEXPORT int
Mpssespi_SafeInit(
    Tcl_Interp *interp)
{
    return TCL_ERROR ;
}
```

```
<<unloading>>=
DLLEXPORT
int
Mpssespi_SafeUnload(
    Tcl_Interp *interp)
{
    return TCL_ERROR ;
}
```

Package Configuration

We also support embedding the package configuration in the package itself. This information will show up the namespace command `pkgconfig`. The `pkgconfig` command has two subcommands, `list` and `get`. This has its basis in TIP 59 and provides a set of name / value pairs of package configuration in formation.

Here we support keys of `pkgname`, `version` and `copyright`.

```
<<static data>>=
static Tcl_Config mpssespi_config[] = {
    {"pkgname", PACKAGE_NAME},
    {"version", PACKAGE_VERSION},
    {"copyright",
"This software is copyrighted 2014 by G. Andrew Mangogna.\
 Terms and conditions for use are distributed with the source code."},
    {NULL, NULL}
} ;
```

We must register the configuration information.

```
<<package configuration>>=
    Tcl_RegisterConfig(interp, PACKAGE_NAME, mpssespi_config, "iso8859-1") ;
```

Source Organization

This document is a literate program. As you have seen it contains both description and source code to the package. The ultimate source for both the PDF documentation and the “C” source code is an ordinary text file formatted as an **asciidoc** document. The process of extracting the source code is conventionally called *tangling*. The tool used tangle this program is called, `atangle`, and is freely available at the mrtools website. The PDF documentation is produced using asciidoc.

This document contains two primary roots, one for the package source and the other for the package tests.

Package Source

The “C” source for the package can be extracted from the root called, `mpssespi.c`.

```
<<mpssespi.c>>=
/*
 * DO NOT EDIT THIS FILE!
 * THIS FILE IS AUTOMATICALLY EXTRACTED FROM A LITERATE PROGRAM SOURCE FILE.
 *
 * This software is copyrighted 2014 by G. Andrew Mangogna.  The following
 * terms apply to all files associated with the software unless explicitly
 * disclaimed in individual files.
 *
 * The author hereby grants permission to use, copy, modify, distribute,
 * and license this software and its documentation for any purpose, provided
```

```
 * that existing copyright notices are retained in all copies and that this
 * notice is included verbatim in any distributions. No written agreement,
 * license, or royalty fee is required for any of the authorized uses.
 * Modifications to this software may be copyrighted by their authors and
 * need not follow the licensing terms described here, provided that the
 * new terms are clearly indicated on the first page of each file where
 * they apply.
 *
 * IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR
 * DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING
 * OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES
 * THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT.  THIS SOFTWARE
 * IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE
 * NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS,
 * OR MODIFICATIONS.
 *
 * GOVERNMENT USE: If you are acquiring this software on behalf of the
 * U.S. government, the Government shall have only "Restricted Rights"
 * in the software and related documentation as defined in the Federal
 * Acquisition Regulations (FARs) in Clause 52.227.19 (c) (2).  If you
 * are acquiring the software on behalf of the Department of Defense,
 * the software shall be classified as "Commercial Computer Software"
 * and the Government shall have only "Restricted Rights" as defined in
 * Clause 252.227-7013 (c) (1) of DFARs.  Notwithstanding the foregoing,
 * the authors grant the U.S. Government and others acting in its behalf
 * permission to use and distribute the software in accordance with the
 * terms specified in this license.
 *
 * *++
 * MODULE:
 *  mpssespi -- Tcl interface to FTDI MPSSE SPI Library
 *
 * ABSTRACT:
 *  This file contains the "C" source for a Tcl interface to libMPSSE,
 *  the FTDI USB to serial converter chip that is capable of emulating
 *  a SPI bus.
 **--
 */

/*
 * INCLUDE FILES
 */
#include <stddef.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
/*
 * We must include libMPSSE_spi.h first as it includes WinTypes.h.
 * There is a conflict in WinTypes.h and tcl.h over the VOID macro and
 * typedef. Ugh!
 */
#include "libMPSSE_spi.h"
#include "tcl.h"

/*
 * MACRO DEFINITIONS
 */
```

```
#ifndef COUNTOF
#   define COUNTOF(a)   (sizeof(a) / sizeof(a[0]))
#endif /* COUNTOF */

<<macro definitions>>

/*
 * TYPE DEFINITIONS
 */
<<type definitions>>

/*
 * EXTERNAL FUNCTION REFERENCES
 */

/*
 * EXTERNAL INLINE FUNCTION REFERENCES
 */

/*
 * STATIC FUNCTION DECLARATIONS
 */

/*
 * EXTERNAL DATA REFERENCES
 */

/*
 * EXTERNAL DATA DEFINITIONS
 */

/*
 * STATIC DATA DEFINITIONS
 */
<<static data>>

/*
 * STATIC FUNCTION DEFINITIONS
 */
<<forward utility functions>>
<<utility functions>>
<<command procedures>>

/*
 * EXTERNAL FUNCTION DEFINITIONS
 */
<<initialization>>

<<unloading>>
```

Test Source

A `tcltest` source file can be extracted starting at the `mpssespi.test` root.

```
<<mpssespi.test>>=
#!/usr/bin/env tclsh
#
# DO NOT EDIT THIS FILE!
# THIS FILE IS AUTOMATICALLY EXTRACTED FROM A LITERATE PROGRAM SOURCE FILE.
#
```

```
# This software is copyrighted 2014 by G. Andrew Mangogna.
# The following terms apply to all files associated with the software unless
# explicitly disclaimed in individual files.
#
# The authors hereby grant permission to use, copy, modify, distribute,
# and license this software and its documentation for any purpose, provided
# that existing copyright notices are retained in all copies and that this
# notice is included verbatim in any distributions. No written agreement,
# license, or royalty fee is required for any of the authorized uses.
# Modifications to this software may be copyrighted by their authors and
# need not follow the licensing terms described here, provided that the
# new terms are clearly indicated on the first page of each file where
# they apply.
#
# IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR
# DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING
# OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES
# THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF
# SUCH DAMAGE.
#
# THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES,
# INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT.  THIS SOFTWARE
# IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE
# NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS,
# OR MODIFICATIONS.
#
# GOVERNMENT USE: If you are acquiring this software on behalf of the
# U.S. government, the Government shall have only "Restricted Rights"
# in the software and related documentation as defined in the Federal
# Acquisition Regulations (FARs) in Clause 52.227.19 (c) (2).  If you
# are acquiring the software on behalf of the Department of Defense,
# the software shall be classified as "Commercial Computer Software"
# and the Government shall have only "Restricted Rights" as defined in
# Clause 252.227-7013 (c) (1) of DFARs.  Notwithstanding the foregoing,
# the authors grant the U.S. Government and others acting in its behalf
# permission to use and distribute the software in accordance with the
# terms specified in this license.
#
#*++
# PROJECT:
#  mpssespi
#
# MODULE:
#  mpssespi.test -- unit tests for mpssespi package
#
# ABSTRACT:
#   This file contains a set of tcltests to exercise the mpssespi package.
#   These tests assume that the package code was built with the "C"
#   preprocessor symbol "TEST" defined.
#*--
#

package require Tcl 8.6
package require cmdline
package require logger
package require platform

# Load the package
set auto_path [linsert $auto_path 0\
        [file normalize [file join .. [platform::identify]]]]
```

```
package forget mpssespi
package require mpssespi

set optlist {
    {level.arg warn {Log debug level}}
}
array set options [::cmdline::getKnownOptions argv $optlist]

::logger::setlevel $options(level)

package require tcltest
tcltest::configure {*}$argv

namespace eval ::mpssespi::test {
    ::logger::initNamespace [namespace current] $::options(level)

    namespace import ::tcltest::*

    log::info "Testing mpssespi version [package require mpssespi]"
    log::notice "These tests assume the package was built with TEST defined"

    <<getNumChannels tests>>
    <<getChannelInfo tests>>
    <<openChannel tests>>
    <<closeChannel tests>>
    <<initChannel tests>>
    <<getChannelConfig tests>>
    <<setChannelConfig tests>>
    <<readChannel tests>>
    <<writeChannel tests>>
    <<readWriteChannel tests>>
    <<isBusy tests>>
    <<changeCS tests>>
    <<writeGPIO tests>>
    <<readGPIO tests>>

    cleanupTests
}

namespace delete ::mpssespi::test
```

Example

The example given below is a transliteration of the example given in the FTDI manual for `libMPSSE-SPI`.

Note

This code **has not been tested**. I don't have access to this particular piece of hardware in a configuration where testing it is possible. However, I have run the example against a SPI bus analyzer and have examined the traces and compared them to the data sheet for the EEPROM part.

A close examination of the "C" code in the FTDI documentation shows that it is not a shining example of "C" coding. The example has been adapted from other uses and some functions are called with arguments that are not used. However, we strive to duplicate the intent of the example below.

First, we must "`require`" the package.

```
<<example.tcl>>=
```

```
#!/usr/bin/env tclsh

package require mpssespi
```

The example defines a couple of functions to read and write the EEPROM. Despite the name of "`read_byte`", it actually reads two bytes. This is because the MicroSemi 94LC56B EEPROM actually transfers data in 16 bit quantities.

Commands codes for the chip that include an address are 3 bits long followed by an 8 bit address. We define a `proc` to do the bit twiddling.

```
<<example.tcl>>=
proc fmtcommand {cmd address} {
    set data [expr {(($cmd & 7) << 5) | (($address >> 3) & 0x1f)}]  ; # ❶
    lappend data [expr {($address & 0x07) << 5}]                    ; # ❷
    return [binary format c* $data]                                 ; # ❸
}
```

❶ The first byte is the 3-bit command and the 5 most significant bits of the address.

❷ The second byte is the remaining 3 least significant bits of the address.

❸ The data that is actually transmitted on the SPI bus is, of course, binary, not Tcl number strings.

```
<<example.tcl>>=
proc read_byte {spichan address} {
    mpssespi writeChannel $spichan [fmtcommand 6 $address]\
            {xferunits bits csdisable false} 12              ; # ❶
    return [mpssespi readChannel $spichan 2 {csenable false}]   ; # ❷
}
```

❶ The whole SPI bus transaction consists of clocking out the write command and address and then reading 16 bits of data. We will accomplish this in two channel calls. So we don't want chip select to be deasserted in between. Also note that we are actually transferring 12 bits. The clocking of the extra bit on a read command effectively discards a dummy bit that the device produces. That just seems to be the way the device works according to the datasheet.

❷ This time, we don't need to enable chip select since we did not disable it on the last command. Since the default channel configuration is to disable chip select at the end of each channel operation, when the read is finished we will actually disable chip select as is required. When transfer options are supplied with a channel read or write command they only affect that particular transfer and the changes are applied against the current channel configuration.

Writing a byte to the EEPROM is more complicated. You must execute a write enable command and, to be safe, execute a write disable command after data is written. The example does this for every byte. A better design might recognize that an entire stream of bytes can be written once writes are enabled. Again we follow the FTDI example here.

```
<<example.tcl>>=
proc write_byte {spichan address data} {
    set wen [binary format c* {0x9f 0xff}]                  ; # ❶
    mpssespi writeChannel $spichan $wen {xferunits bits} 11

    mpssespi writeChannel $spichan [fmtcommand 5 $address]\
            {xferunits bits csdisable false} 11
    mpssespi writeChannel $spichan $data {csenable false}     ; # ❷

    mpssespi writeChannel $spichan {} {xferunits bits csdisable false} 0 ; # ❸
    mpssespi writeChannel $spichan {} {xferunits bits csenable false} 0

    set wen [binary format c* {0x87 0xff}]                  ; # ❹
    mpssespi writeChannel $spichan $wen {xferunits bits} 11
}
```

1. The write enable command is 5 bits, 10011, with don't care (so we set them to 1) for the remaining 6 bits of the 11 bit transaction.

2. The write bus transaction is the 11 bits containing the command and address, followed by two bytes of data. Once again we use two channel commands, making sure not to disable chip select in between.

3. Here the example gets a little vague. There is conditional compilation that is marked as `#if 1` and so we follow that sequence in the code. The chip uses its data out line to indicate the completion status of the write operation. It is necessary to enable chip select and then disable it. One might try the polling data out line using the `isBusy` command and this is also referenced in the example.

4. Finally, we issue the write disable command. Write disable is 5 bits, 10000, with don't care for the other 6 bits of the bus transaction.

Finally, the main program of the example just writes and then reads back 16 address locations.

```
<<example.tcl>>=
proc main {} {
    set numChannels [mpssespi getNumChannels]

    puts "Number of available SPI channels = $numChannels"

    for {set chanNum 0} {$chanNum < $numChannels} {incr chanNum} {
        set info [mpssespi getChannelInfo $chanNum]
        puts "Information on channel number $chanNum:"
        puts "    Type=[format 0x%x [dict get $info type]]"
        puts "    Opened=[dict get $info opened]"
        puts "    Hispeed=[dict get $info hispeed]"
        puts "    Id(vendor)=[format %04x [dict get $info idvendor]]"
        puts "    Id(product)=[format %04x [dict get $info idproduct]]"
        puts "    Location Id=[dict get $info locid]"
        puts "    Serial Number=[dict get $info serialnumber]"
        puts "    Description=\"[dict get $info description]\""
    }

    set spichan [mpssespi openChannel 0]
    mpssespi setChannelConfig $spichan {
        clockrate 5000
        latencytimer 255
        mode 0
        csline DBUS3
        csactive high
    }
    mpssespi initChannel $spichan       ; # (1)

    for {set address 0} {$address < 16} {incr address} {
        set data [expr {$address + 3}]
        puts "writing address $address, data = $data"
        set bindata [binary format S $data]      ; # (2)
        write_byte $spichan $address $bindata
    }

    for {set address 0} {$address < 16} {incr address} {
        set bindata [read_byte $spichan $address]
        binary scan $bindata Su $bindata data    ; # (3)
        puts "reading address $address, data = $data"
    }

    mpssespi closeChannel $spichan
}

# Run the example
```

```
main
```

1. We copy the configuration information from the example. Note this chip uses an active high chip select.
2. The SPI bus deals in raw binary data. The contents of the `data` variable, like all things in Tcl, is represented as a string.
3. Conversely, the data read from the SPI bus is binary and needs to be scanned to obtain a Tcl string representation.

This example is very simple, as most examples are. It also follows the design flow of the original “C” program from FTDI. My suggestion for users of this package is to create chip specific packages that deal with the commands and transaction as required for the specific chip. That chip specific package would then use the `mpssespi` package to perform the actual SPI bus transactions. It’s worth sorting through the chip data sheet and encoding the particular way a chip works into a package so that you can get on with other matters.

Special Linux Considerations

Most modern version of Linux use **udev** as the means of handling hot plugged devices. Associated with **udev** are a set of rules that determine how devices are handled. If you are running a version of Linux that does not use **udev** then this section will not be of any help to you (and you really should be running something more current).

By default when an FTDI device is plugged in, the kernel will load the `usbserial` module and the `ftdi_sio` module. The implicit assumption is that you are going to treat these devices as ordinary serial devices and device special files of the form `ttyUSB?` are created. Unfortunately, this is *not* what you want to happen if you intend to run the `mpssespi` package. The best fix I have been able to find is to manually unload the kernel modules by executing the following commands.

```
sudo rmmod ftdi_sio
sudo rmmod usbserial
```

This will cause the kernel to unload the driver and release the device so that it becomes available for direct access as needed by `libMPSSE-SPI`. I think is should be possible to prevent the drivers from being loaded using **udev** rules, but have been unsuccessful in accomplishing that. Unloading the `ftdi_sio` and `usbserial` kernel modules is a rather draconian approach as it will also disable other FTDI ports that you may wish to have operative.

Another issue that arises is the permissions given to the device special file that is created when the FTDI device is hot plugged. The usual default **udev** rules create the file as owned by `root` and there is no write permission for other users. The best solution for this is to install a `udev` rule. Below is an example rule. This rule matches a particular FTDI device with a known serial number and makes sure the mode of the device special file has write permissions for everyone. Typically this would be installed in the `/etc/udev/rules.d` directory.

```
<<libmpsse-spi.rules>>=
ACTION=="add", SUBSYSTEM=="usb", ATTR{idVendor}=="0403", ATTR{idProduct}=="6010", ATTR{ ↩
    serial}=="FTWSQTDL", MODE="0666"
```

`Udev` rules are very powerful and very flexible. You will have to compose `udev` rules that match your particular usage pattern. For example, leaving out the `ATTR{serial}==...`, clause will cause the rule to match all FTDI devices. You can consult the many posting on the Internet about writing `udev` rules. The example above is just to get you started thinking about what would be needed to make the file permissions work smoothly in your particular configuration.

Another problem you may have when loading the package is occurs if the `D2XX` library is not found. In that case, there will be a core dump with the following message:

```
../../Infra/src/ftdi_infra.c:243:Init_libMPSSE(): NULL expression encountered
```

It may be necessary to set the value of the `LD_LIBRARY_PATH` environment variable to include the directory where the driver is installed. FTDI installation instruction place the driver in `/usr/local/lib`.

Known Problems

As of version 1.0, it seems that invoking the `readWriteChannel` command using transfer units of `bytes` does not function properly. The SPI bus transactions are not correct. However, using the `bits` transfer unit works correctly. This problem requires further investigation.

Building the Package

This package comes with TEA compliant build files. However, the FTDI `libMPSSE-SPI` library is **not** distributed with this package source. To build the package you must obtain the distribution of the library files from FTDI. As of this writing, the `libMPSSE-SPI` library is at version 0.3, released 12 Dec 2011. To run the package it is necessary to have the FTDI `D2XX` driver properly installed on your system.

The `libMPSSE-SPI` zip file may be placed anywhere. The `configure` command to prepare the build accepts a `--with-libmpsse` argument to specify the path to the library files.

The following is the configure command used to build the linux version of the package. This command assumes that `libMPSSE-SPI.zip` was extracted into the same directory as the code for the Tcl package When the `libMPSSE-SPI.zip` file is extracted, it creates a directory named, `Release-SPI`.

```
$ mkdir linux
$ cd linux
$ ../configure --with-tcl=<path to your Tcl installation>\
        --with-libmpsse=../Release-SPI
```

After configuring the software, `make` may be invoked to build it.

Index

C
changeCS, 42
ChangeCSProc, 42
closeChannel, 28
commands
  changeCS, 42
  closeChannel, 28
  getChannelConfig, 14
  getChannelInfo, 7
  getNumChannels, 6
  initChannel, 12
  isBusy, 40
  openChannel, 11
  readChannel, 30
  readGPIO, 45
  readWriteChannel, 38
  setChannelConfig, 20
  writeChannel, 34
  writeGPIO, 43
ConvertXferUnitsToBytes, 34

D
DeleteHandleMapping, 5
DeleteMPSSEPkgInfo, 3

F
functions
  ChangeCSProc, 42
  ConvertXferUnitsToBytes, 34
  DeleteHandleMapping, 5
  DeleteMPSSEPkgInfo, 3
  GetChannelConfigProc, 15
  GetDirectionBitMask, 27
  GetDirectionDict, 19
  GetNumChannelsProc, 6
  GetPinValueDict, 20
  InitChannelProc, 13
  IsBusyProc, 41
  LookUpHandle, 5
  Mpssespi_Init, 48
  Mpssespi_SafeInit, 49
  Mpssespi_Unload, 49
  NewHandleMapping, 3
  NewMPSSEPkgInfo, 2
  OpenChannelProc, 11
  ReadGPIOProc, 45
  ReadWriteChannelProc, 38
  SetStatusResult, 47
  SetTransferOptions, 33
  VerifyOutputLength, 37
  WriteChannelProc, 35
  WriteGPIOProc, 44

G
getChannelConfig, 14
GetChannelConfigProc, 15
getChannelInfo, 7
GetDirectionBitMask, 27
GetDirectionDict, 19
getNumChannels, 6
GetNumChannelsProc, 6
GetPinValueDict, 20

I
initChannel, 12
InitChannelProc, 13
isBusy, 40
IsBusyProc, 41

L
LookUpHandle, 5

M
Mpssespi_Init, 48
Mpssespi_SafeInit, 49
Mpssespi_Unload, 49

N
NewHandleMapping, 3
NewMPSSEPkgInfo, 2

O
openChannel, 11
OpenChannelProc, 11

R
readChannel, 30
readGPIO, 45
ReadGPIOProc, 45
readWriteChannel, 38
ReadWriteChannelProc, 38

S
setChannelConfig, 20
SetStatusResult, 47
SetTransferOptions, 33

V
VerifyOutputLength, 37

W
writeChannel, 34
WriteChannelProc, 35
writeGPIO, 43
WriteGPIOProc, 44