

Developing Reusable Device Drivers for MCU's

By Jacob Beningo

www.beningo.com

<http://www.linkedin.com/in/jacobbeningo>

twitter : Jacob_Benigo

EDN Blog - Embedded Basics:

<http://www.edn.com/electronics-blogs/4375394/Embedded-Basics>



Developing Reusable Device Drivers for MCU's

July 31, 2012

Rev A

Embedded Systems Conference – East 2012

Page 2 of 20

Table of Contents

INTRODUCTION	3
DRIVER CODE ORGANIZATION	3
APPLICATION PROGRAMMING INTERFACE (API'S).....	4
POINTER ARRAYS.....	5
CONFIGURATION TABLES	6
DIGITAL INPUT/OUTPUT DRIVER DESIGN	8
SERIAL PERIPHERAL INTERFACE (SPI) DRIVER DESIGN	13
CONCLUSION	20

Introduction

The rate at which society expects products to be released and refreshed has been steadily decreasing over the last two decades. The result has left development teams scrambling to implement the most basic product features before the product launch date. Designing a new product from scratch takes time, effort and money that is often unavailable.

Embedded software developers often look to chip manufacturers to provide example code and drivers for their processors to help accelerate the design cycle. Unfortunately the provided code often lacks a layered architecture that would allow the code to be easily reused. In addition, the code is often sparingly documented which makes fully understanding what is being done difficult. The result is poorly crafted code that is difficult to read and comprehend that offers no possibility of reuse with the next product. Time and effort is forced to focus on developing low level drivers rather than on implementing the product features at hand.

This paper will explore methods and techniques that can be used to develop reusable abstracted device drivers that will result in a sped up development cycle. A method for driver abstraction is examined in addition to a brief look at key C language features. A layered approach to software design will be explored with common driver design patterns for Timers, I/O, and SPI which can then be expanded upon to develop drivers for additional peripherals across a wide range of processor platforms.

Driver Code Organization

There are many different ways in which software can be organized. In fact, nearly every engineer has their own opinion on how things should be done. In this paper, with the intention of creating reusable drivers and reusable design patterns, the software will be broken up into layers which will include driver and application layers. The primary focus will be on the driver layer with the intent that the same basic principles can be applied to higher layers.

The driver layer will consist of peripheral interface code as one would expect; however, the drivers will attempt to remain generic to the peripheral. This will allow them to be used and configured for any range of applications. The driver layer can be compiled into a separate library that can then be dropped into any project. The configuration for each driver would be contained within configuration modules that would be part of its own layer. Each application can then uniquely configure the driver and application layers to match the requirements. Figure 1 shows how the configuration and driver code would be organized.

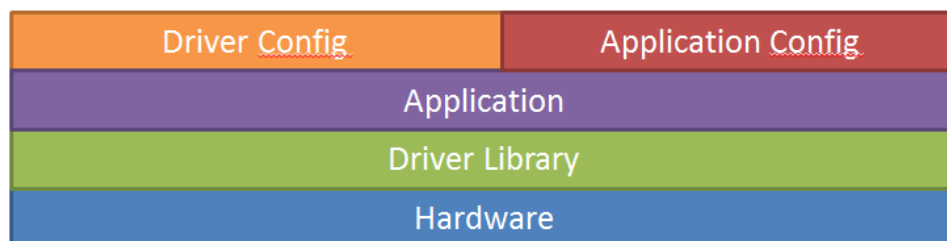


Figure 1 – Layered Organization

Application Programming Interface (API's)

One of the most critical steps in developing a reusable driver framework is to define the Application Programming Interface (API's). Properly defining the API's allows for a standard interface to be used to access hardware across multiple platforms and projects. This is something that high level operating systems have done relatively well over the years.

There are many possible ways in which these API's can be defined and is often dictated by programmer preferences. For this reason, the developed API's should become part of the development teams' software coding standard. The end goal is to define the API's in a way that meets general requirements of the system but allows the power of each peripheral to be fully utilized.

There are software API's available that can provide a starting point. It is possible to adopt formats used by the Linux kernel, Arduino libraries, AUTOSAR, or a custom driver API that is a mix. It really doesn't matter provided that the format is well documented and used across all platforms and projects.

It is useful to define the API's for common and useful features for each of the peripherals. Each peripheral will require an initialization function in addition to functions that allow the peripheral to perform its functions. For example, Listing 1 shows a possible interface for a Digital Input/Output driver. It consists of an initialization function, a read, write and toggle function.

```
void Dio_Init(const Dio_ConfigType *Config);  
BOOL Dio_ReadChannel(Dio_ChannelType Channel);  
void Dio_WriteChannel(Dio_ChannelType Channel, BOOL State);  
void Dio_ToggleChannel(Dio_ChannelType Channel);
```

Listing 1 – Digital Input/Output API

The Serial Peripheral Interface (SPI) and EEPROM API's can be found below in Listing 2 and Listing 3. These are the example interfaces that will be used in this paper.

```
void Spi_Init(const Spi_ConfigType *config);  
void Spi_Transfer(const Spi_TransferType *config);
```

Listing 2 – Serial Peripheral Interface API

```
void Eep_Init(const Spi_TransferType *config);  
void Eep_Read(uint8 *Dest, uint32 Src, uint32 Size);  
void Eep_Write(uint32 Dest, uint8 *Src, uint16 Size);  
void Eep_PageErase(uint32 Dest);
```

Listing 3 – EEPROM API

In these examples the coding standard typically uses a three letter designation to indicate the peripheral or board support interface followed by a single underscore. The underscore precedes the interface function. Each word is capitalized in order to ease readability of the code.

It should be noted that uint8, uint16 and uint32 are respectively uint8_t, uint16_t and uint32_t. The author has found that it is fairly obvious what these types are and continually writing _t after every type doesn't have any added value. This is open to personal interpretation but is the convention that will be used throughout the rest of this paper.

Pointer Arrays

One of the fundamental issues in driver design is deciding how to map to the peripheral registers. Over the years there have been many different methods that have been used such as setting up structures to define bit maps or simply writing the desired value to the register; however, my all-time favorite method is to create an array of pointers that map to the peripheral registers. This method offers an elegant way to group peripheral registers into logical channels and provide a simple method to not only initialize the peripheral but also access its data.

The pointer array method is easily ported and can be used to create standard API's and application code that can work across different hardware platforms, allowing for application code to be shared. If properly written, it also creates code that is far easier to read and understand which makes software maintenance easier.

The concepts of pointer arrays are a relatively straight forward method for mapping to a peripheral. The idea is to create an array where each index of an array is a pointer to a peripheral register of a particular type. For example, for a microcontroller with multiple GPIO ports, a pointer array would be setup to access the direction registers of each of the available ports (Listing 4). Another pointer array would be setup to access the input and output registers. Each register type would be associated with its own pointer array.

```
/**
 * Defines a table of pointers to the ports data direction register on the
 * microcontroller.
 */
uint16 volatile * const portsddr[NUM_PORTS] =
{
    (uint16*)&TRISB, (uint16*)&TRISC, (uint16*)&TRISD,
    (uint16*)&TRISE, (uint16*)&TRISF, (uint16*)&TRISG
};
```

Listing 4: Pointer Array for GPIO

It is important to take note of the way in which the pointer array is declared. The pointer array portsddr is a constant pointer to a volatile uint16. Notice that the declaration is defined from right to left. The pointer to the register is a constant pointer but declaring it as a volatile uint16 notifies the compiler that the value that is being pointed to may change on its own without interaction from the software.

There are many advantages to using this approach to memory mapping. First, it allows registers of the same function to be logically grouped together. This allows the software engineer to view each peripheral as a separate channel of the MCU. For example, timer 1 and timer 2 could be looked at as being two different timer channels.

To setup the period register of each timer would only require a simple write to the proper channel index of the period pointer array. The index of the pointer array then becomes a channel access index. For instance, pointer array index 0 would be associated with Timer 1; pointer array index 1 would be associated with Timer 2.

Next, when the peripherals start to look like channels, it becomes easy to create an abstract method of not only initializing but also of accessing each of the peripheral data. This allows a simple loop to be used to initialize each peripheral (Listing 5). It allows the data of the peripheral to be accessed by simply using the correct channel index. This results in a driver framework that is not only easy to understand and reuse but a framework that abstracts the device registers.

```
// Initialize each channel to zero.  
for(i=0; i < NUM_TIMERS; i++)  
{  
    *tmrreg[i] = 0;           //Clear timer register  
}
```

Listing 5: Timer Initialization Loop

Finally, it allows the developer to create configuration tables for each peripheral. Instead of always writing custom initialization code, the developer can create a reusable driver that takes the configuration table as a parameter. The initialization function then loops through the table one channel at a time and initializes the peripheral registers through the pointer array. This allows the driver to become a library module that is tested time and time again resulting in proven code that can accelerate the next project.

Configuration Tables

Memory mapping microcontroller peripherals using pointer arrays allows the peripheral to be viewed as a collection of channels that can be configured through an index in a loop. By taking this generic approach to memory mapping, a technique is needed to control exactly what is put into the registers. Configuration tables serve as a useful tool for this exact purpose.

A configuration table is exactly what it sounds like; it is a collection of channels and values that are used to configure a peripheral. The most useful way in which to define a configuration table is to create a typedef structure that contains all of the needed fields to setup each channel. Start by examining the peripheral registers of interest. For example, examining the timer peripheral may result in determining that the necessary fields that should be included in the configuration table are channel, period and control fields. The table elements can then be defined by the structure shown in Listing 6.

```
/**
 * Defines the timer configuration table elements that are used
 * by Tmr_Init to configure the timer registers.
 */
typedef struct
{
    uint8 TimerChannel;      /**< Name of Timer */
    uint8 volatile Period;   /**< Period register value */
    uint8 volatile Control;  /**< Timer Control register value */
}Tmr_ConfigType;
```

Listing 6: Configuration Table Definition

The Tmr_ConfigType defines all of the data that is required in order to setup a single timer peripheral. Since most microcontrollers contain more than a single timer, an array of Tmr_ConfigType would be created with each index of the array representing a channel (a single timer module). Before a configuration table can be defined it is useful to first define channel types for the table. The channel will be used to access indices in an array that belong to that channel and in turn will allow the application code to manipulate that particular timer.

```
/**
 * This enumeration is a list of the timer channels
 */
typedef enum
{
    TIMER1,      /**< Timer 1 */
    TIMER2,      /**< Timer 2 */
    NUM_TIMERS   /**< Number of timers on the microcontroller */
}Tmr_ChannelType;
```

Listing 7: Timer Channel Definitions

In Listing 7, a typedef enumeration is used to create the channel names. Since enumerations start at 0 (in C anyways), TIMER1 can be used to access index 0 of an array that contains information about TIMER1. NUM_TIMERS then holds the value for the number of timers that are available. This can be used in the driver initialization to loop through and configure each channel up to NUM_TIMERS.

Once the channel type has been defined it is possible to fill in the configuration table with the values that will be used to configure the timers. Listing 8 shows an example configuration table based on the Tmr_ConfigType structure. The configuration table is defined as a const since the configuration data will not be changing during run-time. This will allow the configuration tables to remain in flash and not take up valuable space in RAM. Each channel is listed along with a period and a control register value. If a clock module were developed, it would be possible to use a time in microseconds instead of a period. The timer module would then use the clock module to properly calculate what the period register should be.

```
/**
 * This configuration table is used to configure the behavior and function of
 * the timers.
 */
const Tmr_ConfigType Tmr_Config[] =
{
    //      Timer          Timer          Control Register
    //      Channel        Period          Attributes
    //
    {TIMER1,      20000,      0,          },
    {TIMER2,      2000,      0,          }
};
```

Listing 8: Configuration Table Example for 2 timers

If Listing 8 was being used within a real project, the period values would correspond to the number of ticks of the timer required before an interrupt or some other useful system event would occur. The control register attributes would be representative of other registers that would require setup. It would be possible to include enabling and disabling of interrupts for each timer in addition to controlling the interrupt priority. Items included in the configuration table may vary from peripheral to peripheral based on what features are supported by the manufacturer. The process, design pattern and look of each table would be similar and familiar leaving little guess work as to how the module is configured.

Digital Input/Output Driver Design

General Purpose Input / Output or Digital Input / Output are one of the most fundamental peripherals on every microcontroller. However, in most applications figuring out how the devices pins are configured can be a nightmare. They are usually configured as shown in Listing 9 except that instead of only displaying four registers there are hundreds of them! This type of definition was fine when devices only had 8 bit ports and there were only one or two per device. However, today microcontrollers can have 100's of pins which need to be configured. This is why we are going to examine an approach to pin mapping using arrays of pointers. You will find at the end of this section that this method proves far easier to determine the configuration of a pin once the work has been put in up front.

```
TRISA = 0x42;
TRISB = 0x71;
PORTA = 0x05;
PORTB = 0x14;
```

Listing 9: Example I/O Configuration

The first step that should be performed when developing the digital input / output driver is that the device registers should be examined in the datasheet. While there are common features across manufacturers and chip families, features do vary.

Next, write down a list of all the features that should be implemented in the driver. Some example features for a digital input / output driver are pin direction, initial state and the function

that the pin will serve such as GPIO, SPI, PWM, etc. Once this list has been compiled it can be put into a configuration structure as shown in Listing 10.

```
/**
 * Defines the digital input/output configuration table elements that are used
 * by Dio_Init to configure the Dio peripheral.
 */
typedef struct {
    uint8 Channel;           /**< The I/O channel */
    uint8 PinType            :2; /**< Set Pin Type - ANALOG, DIGITAL,etc */
    uint8 Direction          :1; /**< Data Direction - OUTPUT or INPUT */
    uint8 Data               :1; /**< Data - HIGH or LOW */
    uint8 Function           :2; /**< Mux Function - Dio_Peri_Select */
}Dio_ConfigType;
```

Listing 10: Digital I/O Configuration Structure

With the list of configuration parameters developed, the only piece missing before the table can be filled in is the channel definitions. These definitions can start out as a generic list such as PORTA_0, PORTA_1 etc; however, once in an application it is far more convenient to label the channels with useful designations. For example, LED_RED, LED_BLUE would replace the generic label so that the developer knows exactly what output is being manipulated. An example channel definition can be found in Listing 11 as a typedef enumeration.

```
/**
 * This enumeration is a list of the general purpose I/O pin channels. They
 * are used by the by the Dio functions for reading and writing to the
 * digital pins.
 */
typedef enum
{
    LED_RED,           /**< PORTA_0 */
    EEP_CS,            /**< PORTA_1 */
    MOSI,              /**< PORTA_2 */
    MISC,              /**< PORTA_3 */
    CLK,               /**< PORTA_4 */
    PORTA_5,           /**< PORTA_5 */
    PORTA_6,           /**< PORTA_6 */
    PORTA_7,           /**< PORTA_7 */
    NUM_DIGITAL_PINS  /**< Number of digital pins */
}Dio_ChannelType;
```

Listing 11: Digital I/O Channel Types

Once the channels have been defined it is straightforward to generate the configuration table. Create a const array that is of type Dio_ConfigType and then start populating how each channel (pin) should be configured. For instance, for the LED_RED channel, the pin should be configured as a digital pin, with the direction of OUTPUT and an initial state of HIGH. The pin function would of course be set to GPIO. A complete example of the configuration table can be seen in Listing 12.

```
/**
 * This configuration table is used to configure the behavior and function of
 * the digital i/o. The channels are defined in dio_cfg.h. The configuration
 * consists of Pin Type (Analog or Digital), Direction (INPUT or OUTPUT), Initial
 * pin state (LOW or HIGH), Function (GPIO, SPI, etc)
 */
const Dio_ConfigType Dio_Config[] =
{
//
// Channel          Pin Type      Direction      Initial      Function
// State
//
{LED_RED,          DIGITAL,      OUTPUT,        HIGH,        GPIO,    }, //PORTA_0
{EEP_CS,           DIGITAL,      OUTPUT,        HIGH,        GPIO,    }, //PORTA_1
{MOSI,            DIGITAL,      OUTPUT,        HIGH,        SPI,     }, //PORTA_2
{MISO,            DIGITAL,      INPUT,         HIGH,        SPI,     }, //PORTA_3
{CLK,             DIGITAL,      OUTPUT,        HIGH,        SPI,     }, //PORTA_4
{PORTA_5,          ANALOG,       OUTPUT,        LOW,         GPIO,    }, //PORTA_5
{PORTA_6,          DIGITAL,      OUTPUT,        LOW,         GPIO,    }, //PORTA_6
{PORTA_7,          DIGITAL,      OUTPUT,        LOW,         GPIO,    } | //PORTA_7
};
```

Listing 12: Digital I/O Configuration Table example

With the configuration table and channels defined the next step in the process of developing a digital input / output driver is to memory map the peripheral registers to a pointer array. Once this is done the initialization function can be developed. As a simple example, the code in Listing 13 assumes that the device is a single port device. The digital input register, digital direction register, and output state register are all mapped. The final piece of code creates an array that allows for the driver to access an individual bit within a register based on the pin number. For example, pin 3 would be accessed by bit 2 in a register which is a 1 shifted to the left by 2. The code can be simplified in the initialization function if these bit shifts are stored in an array.

```
/**
 * Defines a table of pointers to the ports on the microcontroller.
 */
uint16 volatile * const portsin[NUM_PORTS] =
{
    (uint16*)&PORTA
};

/**
 * Defines a table of pointers to the ports data direction register on the
 * microcontroller.
 */
uint16 volatile * const portssdr[NUM_PORTS] =
{
    (uint16*)&TRISA
};

/**
 * Defines a table of pointers to the ports latch register on the
 * microcontroller
 */
uint16 volatile * const ports[NUM_PORTS] =
{
    (uint16*)&LATA
};

/**
 * Defines a table of pins for the microcontroller.
 */
const uint16 pins[NUM_PINS_PER_PORT] =
{
    (1UL<<0), (1UL<<1), (1UL<<2), (1UL<<3), (1UL<<4), (1UL<<5), (1UL<<6), (1UL<<7)
};
```

Listing 13: Pointer Array Memory Maps for Digital I/O

After a good amount of preparation the initialization function is finally ready to be written. It is relatively simple. A pointer to the configuration table is passed to the function. A simple loop is used to setup each of the pins. During each pass, each configuration value is read and based on the value a register is configured. Listing 14 shows how each of the configuration values is recorded in the registers. As you can see this code is straight forward and easily re-used. The only change is that the pointer array would need to be updated for the correct registers. Minor changes to how the analog pins are configured may be necessary but as long as the API is followed application code can be reused from one processor to the next.

```
/******
 * Function : Dio_Init()
 */**
 * \section Description Description:
 *
 * This function is used to initialize the Dio based on the configuration table
 * defined in dio_cfg module.
 *
 * \param - const Dio_ConfigType * Config - pointer to the config table.
 *
 * \return None.
 *
 *****/
void Dio_Init(const Dio_ConfigType * Config)
{
    uint8 i = 0;
    uint8 number = 0;           // Port Number
    uint8 position = 0;         // Pin Number

    //Loop through all pins, set the data register bit and the data direction
    //register bit according to the dio configuration table values
    for (i = 0; i < NUM_DIGITAL_PINS; i++)
    {
        number = Config[i].Channel / NUM_PINS_PER_PORT;
        position = Config[i].Channel % NUM_PINS_PER_PORT;

        // Set the AN pins as analog or digital
        if (Config[i].PinType == ANALOG)
        {
            AD1PCFGL &= ~pins[position];
        }
        else if (Config[i].PinType == DIGITAL)
        {
            AD1PCFGL |= pins[position];
        }

        // Set the Data register bit for this channel
        if (Config[i].Data == HIGH)
        {
            *ports[number] |= pins[position];
        }
        else
        {
            *ports[number] &= ~pins[position];
        }

        // Set the Data Direction register bit for this channel
        if (Config[i].Direction == OUTPUT)
        {
            *portsddr[number] &= ~ pins[position];
        }
        else
        {
            *portsddr[number] |= pins[position];
        }
    }
}
```

Listing 14: Example Digital I/O Initialization Function

A quick example of how to write an additional function would be useful. In many applications it is often useful to toggle an led in order to see that the system is functioning. Listing 15 demonstrates how to access the pointer array to toggle a given channel.

```

/*****
 * Function : Dio_ToggleChannel()
 ***/
 * \section Description Description:
 *
 * This function is used to toggle the value of the specified digital i/o.
 *
 * \param - uint8 Channel - the pin identifier.
 *
 * \return None.
 *
 *****/
void Dio_ToggleChannel(Dio_ChannelType Channel)
{
    *ports[Channel/NUM_PINS_PER_PORT] ^= (1UL<<(Channel%NUM_PINS_PER_PORT));
}

```

Listing 15: Digital I/O Driver Definition

The usage for this function is very straight forward. Simply pass one of the DioChannelType channels such as LED_RED. The function could be called at a rate of 500 ms. Listing 16 demonstrates how other functions can be used along with the Dio_ToggleChannel.

```

// Toggle the Red LED
Dio_ToggleChannel(RED_LED);

// Turn on the Red LED
Dio_WriteChannel(RED_LED, ON);

// Turn off the Red LED
Dio_WriteChannel(RED_LED, OFF);

```

Listing 16: Digital I/O Functions

Serial Peripheral Interface (SPI) Driver Design

The serial peripheral interface (SPI) is a commonly used interface. It consists of three communication lines in addition to a chip select line. It is often used to communicate with EEPROM, SD cards and many other peripheral devices. Most SPI interfaces can reach speeds in excess of 4 Mbps.

Just like with the Digital I/O driver, the first step to develop a SPI driver will be to develop the configuration table. An example configuration structure can be found in Listing 16.

```
/**
 * Defines the configuration data required to initialize the SPI peripheral.
 */
typedef struct
{
    uint8 ChannelName;      /**< defines the name of the SPI channel          */
    uint8 SpiEnable;        /**< defines the whether the SPI channel is enabled      */
    uint8 Master_Mode;      /**< defines the peripheral Master/Slave mode          */
    uint16 BaudRate;        /**< defines the baud rate                                */
    uint8 CommSelect;       /**< defines the Word/Byte communication select bit      */
}Spi_ConfigType;
```

Listing 16: SPI Configuration Table Definitions

Depending on the part that is being used, there may be more than a single SPI channel per chip. In Listing 17, a Spi_ChannelType enumeration is declared that defines the possible SPI channels. As before, these channels can be used to access the pointer arrays in addition to controlling the behavior of the application.

```
/**
 * This enumeration defines a list of the spi channels
 */
typedef enum
{
    SPI_1,          /**< SPI 1 */
    SPI_2,          /**< SPI 2 */
    NUM_SPI_CHANNELS /**< Number of SPI channels */
}Spi_ChannelType;
```

Listing 17: SPI Channel Definitions

There are a number of features that are common to SPI peripherals that are configured by the configuration table. SPI offers the ability for the processor to behave as a Master which controls the communication with a slave device. It also allows the processor to be configured as the slave device. If there is more than a single SPI channel, each channels baud rate can be individually configured and the width of each communication data chunk.

Listing 18 shows how a two channel SPI processor could be configured. In this example, the first SPI peripheral is enabled during start-up as a master device with a baud rate of four Mbps. Each communication with a slave device occurs in byte communication. The second channel is disabled at start-up but if it were enabled during operation it would act as a slave device. A slave device requires a chip select to be enabled in order to clock in data. The slave channel would be configured to expect a baud rate of 400 kbps and receive the data in 2 byte data chunks.

```
*****/
/**
 * This configuration table is used to configure the behavior and function of
 * the spi channels. The channels are defined in spi_cfg.h.
 *
 *****/
const Spi_ConfigType Spi_Config[] =
{
//      SPI      Channel      Communication
// Channel  Enable   Master Mode   Baud Rate   Selection
//
    {SPI_1,    ENABLED,    MASTER,    SPI_BAUD_4M,    BYTE_WIDE},
    {SPI_2,    DISABLED,   SLAVE,     SPI_BAUD_400K,  BYTE_BYTE}
};
```

Listing 18: SPI Configuration Table Example

There are only a couple of functions that are really necessary in order to get a SPI driver up and running. The first is the initialization function. The second is a transfer function that sends out and receives data. The Spi_Init function would accept a pointer to the configuration table. The Spi_Transfer function would also accept a pointer to a configuration table. Listing 19 shows the prototypes for these functions.

```
void Spi_Init(const Spi_ConfigType *config);
void Spi_Transfer(const Spi_TransferType *config);
```

Listing 19: SPI Function Prototypes

There is a major difference between the two configuration tables that each of these functions take for parameters. The Spi_Init configuration initializes the peripheral from a general standpoint; for example, peripheral baud rate. The Spi_Transfer configuration describes how a particular device will communicate over SPI. For example, two different SPI slave devices may be setup to communicate in completely different ways. One may be active low chip select with a certain phase and clock polarity while another device may be completely opposite. In this case, Spi_Transfer allows each device to setup with the same SPI channel and each individual data transfer to be configured as required. Listing 20 shows some examples of what might be found in the configuration structure.

The Spi_Init function would be written in the same manner as the digital input/output initialization. Pointer arrays would be declared and the initialization would loop through each individual channel, setting up the registers per the configuration table. The Spi_Transfer function is far more interesting to take a look at. It consists of a number of steps to properly send data.

The first step of the Spi_Transfer function is to configure the SPI peripheral for the communication. This is usually done by first resetting the peripheral. The purpose of this is to clear out any old transfer data and prepare the peripheral for new configuration data. Next, the clock phase and polarity are configured. The transfer mode (Master or Slave) is setup prior to enabling the SPI peripheral. This can be seen in Listing 21.

```
/**
 * The SPI Transfer Type structure is used to set the configuration for
 * transmitting SPI data.
 */
typedef struct
{
    uint8 SpiChannel;          /**< The spi channel to be used */
    uint8 ChipSelect;          /**< The dio channel to be used for CS */
    uint8 Cs_Polarity;         /**< The active state of CS */
    uint16 NumBytes;           /**< The number of bytes to send */
    uint8 *TxRxData;           /**< Pointer to the data to transfer */
    uint8 Polarity:1;          /**< Transfer data polarity */
    uint8 Phase:1;             /**< Transfer data phase */
    uint8 Direction:1;         /**< Bit direction */
}Spi_TransferType;
```

Listing 20: Spi Transfer Configuration

At this point the peripheral is configured and ready to send data. In this example, the SPI is configured as a master. This means that the processor controls the communication on the bus. In order to talk to a slave device, the chip select of the device needs to be toggled to tell it to prepare to receive data. Chip selects can be either active high or active low. The configuration data is used to determine which is correct to communicate with this slave device and the chip select is set active. Listing 22 shows an example function that can be used to set a slave device into active mode. Listing 23 shows the opposite function used to put the slave in an inactive state.


```

/*****
 * Function : Spi_Setup()
 **/
 * \section Description Description:
 *
 * This function is used to configure the SPI peripheral to communicate with
 * a particular slave device.
 *
 * \param -   const Spi_TransferType * config
 *
 * \return   None.
 *
 *****/
inline void Spi_Setup(const Spi_TransferType * config)
{
    //Reset the module. This disables SPI and clears any flags but retains any
    //current register settings for the SPI peripheral.
    *spistat[config->SpiChannel] &= ~REGBIT15;

    /*****
     * Set the polarity, phase, and shifter direction (LSBit first or MSBit first)
     * based on the configuration. Set for master mode and enable the SPI.
     * Disable SPI Interrupts.
     *****/
    //Set the spi channel polarity
    if(config->Polarity == POLARITY_HIGH)
    {
        *spicon1[config->SpiChannel] |= REGBIT6;
    }
    else
    {
        *spicon1[config->SpiChannel] &= ~REGBIT6;
    }

    //Set the spi channel phase
    if(config->Phase == PHASE_HIGH)
    {
        *spicon1[config->SpiChannel] |= REGBIT9;
    }
    else
    {
        *spicon1[config->SpiChannel] &= ~REGBIT9;
    }

    *spicon1[config->SpiChannel] |= REGBIT5;           //Set SPI channel to master mode
    *spistat[config->SpiChannel] |= REGBIT15;          //Enable the spi module

    (void)*spibuf[config->SpiChannel]; // Perform dummy read to clear the buffer
}

```

Listing 21: SPI Transfer Peripheral Setup Function

```
/* *****
 * Function : Spi_SetCs()
 */
 * \section Description Description:
 *
 * This function is used to select a slave device. It toggles an I/O line
 * into the active state.
 *
 * \param - const Spi_TransferType * config
 *
 * \return None.
 *
 */
inline Spi_SetCs(const Spi_TransferType * config)
{
    //Select the device
    if(config->Cs_Polarity == CS_ACTIVE_LOW)
    {
        Dio_WriteChannel(config->ChipSelect, LOW);
    }
    else
    {
        Dio_WriteChannel(config->ChipSelect, HIGH);
    }
}
```

Listing 22: SPI Slave Chip Select Active Function

```
/* *****
 * Function : Spi_ClearCs()
 */
 * \section Description Description:
 *
 * This function is used to de-select a slave device. It toggles an I/O line
 * into the inactive state.
 *
 * \param - const Spi_TransferType * config
 *
 * \return None.
 *
 */
inline Spi_ClearCs(const Spi_TransferType * config)
{
    // Latch the data into the slave by de-selecting the chip select.
    if(config->Cs_Polarity == CS_ACTIVE_LOW)
    {
        Dio_WriteChannel(config->ChipSelect, HIGH);
    }
    else
    {
        Dio_WriteChannel(config->ChipSelect, LOW);
    }
}
```

Listing 23: SPI Slave Chip Select Inactive Function

The data is then transferred one chunk at a time to the slave device; however, before data is transferred the order of the bytes and bits needs to be set. Some devices expect data LSB to MSB while others MSB to LSB. This is part of the configuration. If required the Spi_Transfer function reorders the bytes and transmits them. At the end of each chunk out of data, a new chunk of data is read in. Once all of the data has been transmitted, the chip select is cleared and the data transfer is complete. The final Spi_Transfer function can be found in Listing 24.

```

/*****
 * Function : Spi_Transfer()
 ***/
 * \section Description Description:
 *
 * This function is used to transfer data through the SPI peripheral.
 *
 * \param - const Spi_TransferType * config
 *
 * \return None.
 *
 *****/
void Spi_Transfer(const Spi_TransferType * config)
{
    uint8 i = 0;                // loop index (ranges from 0 to NumBytes)
    uint8 j = 0;                // data pointer index

    // Setup the spi registers with the spi devices communication settings
    Spi_Setup(config);

    // Initialize the Chip Select
    Spi_ChipSelect(config);

    /*****
     * Transmit (and receive) the data
     *****/
    for(i = 0; i < config->NumBytes; i++)
    {
        // Check the shift direction. If it is LSB first then reverse the order
        if (config->Direction == 1)
        {
            j = config->NumBytes - i - 1; // LSB first selected. Reverse the index.
        }
        else
        {
            j = i;                        // MSBit first selected. Normal index.
        }

        // Transmit the data to the slave device.
        *spibuf[config->SpiChannel] = (*(config->TxRxData + j));

        // Wait for the transfer to complete then read the data from the slave device
        while((*spistat[config->SpiChannel] & REGBIT0) == 0);
        *(config->TxRxData + j) = *spibuf[config->SpiChannel];
    } // End for

    // Clear the chip select since the data transfer is complete
    Spi_ClearCs(config);
}

```

Listing 24: SPI Transfer Function Example

Conclusion

There are many methods that can be used to develop device drivers. Using pointer arrays with configuration tables opens up the possibility of developing reusable drivers that follow a design pattern that can be used across not only families of processors but across platforms as well. Following these simple design patterns will drastically speed up the driver design cycle, leaving more time for focusing on the application challenges rather than low level chip functions.

Keeping to a common set of driver API's allows higher level application code to be easily ported from one project to the next. This continues the trend of speeding up the design cycle while increasing the quality of the components used.