**Online from** [**http://www.embedded.com/design/mcus-processors-and-socs/4420068/Building-reusable-device-drivers-for-microcontrollers#**](http://www.embedded.com/design/mcus-processors-and-socs/4420068/Building-reusable-device-drivers-for-microcontrollers#)

**Building reusable device drivers for microcontrollers**

**Jacob Beningo** - January 10, 2013

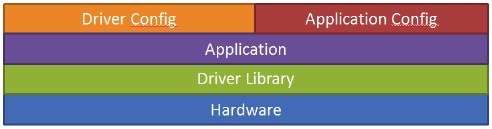
The rate at which society expects products to be released and refreshed has been steadily decreasing over the last two decades. The result has left development teams scrambling to implement the most basic product features before the product launch date. Designing a new product from scratch takes time, 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 â€“ Layered Organization**

**Application Programming Interface (APIs)**  
One of the most critical steps in developing a reusable driver framework is to define the Application Programming Interface (APIs). Properly defining the APIs 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 APIs can be defined and is often dictated by programmer preferences. For this reason, the developed APIs should become part of the development teams software coding standard. The end goal is to define the APIs in a way that meets general requirements of the system but allows the power of each peripheral to be fully utilized.

There are software APIs 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 APIs 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.

http://m.eet.com/media/1175342/jbdrv02.jpg  
**Listing 1 “ Digital Input/Output API**

**Page 2**

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.

http://m.eet.com/media/1175343/jbdrv03.jpg  
**Listing 2 – Serial Peripheral Interface API**

http://m.eet.com/media/1175344/jbdrv04.jpg  
**Listing 3 – EEPROM API**

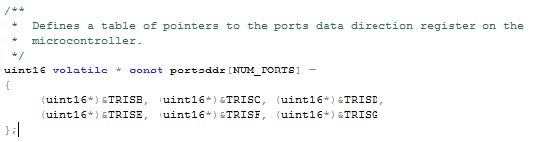
In these examples the coding standard typically uses a three letter designation to indicate the peripheral or board support interface followed by a single underscore. The underscore precedes the interface function. Each word is capitalized in order to ease readability of the  
code.

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.

  
**Listing 4: Pointer Array for GPIO**

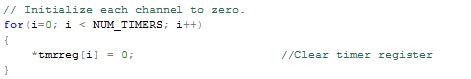
It is important to take note of the way in which the pointer array is declared. The pointer array portsddr is a constant pointer to a volatile uint16. Notice that the declaration is defined from right to left. The pointer to the register is a constant pointer but declaring it as a volatile uint16 notifies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

**Page 3**

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.

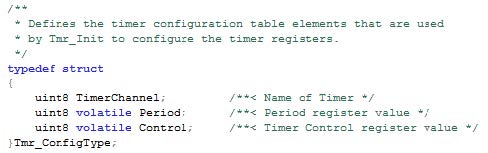
Next, when the peripherals start to look like channels, it becomes easy to create an abstract method of not only initializing but also of accessing each of the peripheral data. This allows a simple loop to be used to initialize each peripheral (Listing 5). It allows the data of the peripheral to be accessed 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.

  
**Listing 5: Timer Initialization Loop**

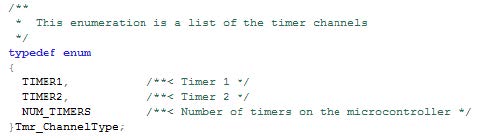
Finally, it allows the developer to create configuration tables for each peripheral. Instead of always writing custom initialization code, the developer can create a reusable driver that takes the configuration table as a parameter. The initialization function then loops through the table one channel 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.

  
**Listing 6: Configuration Table Definition**

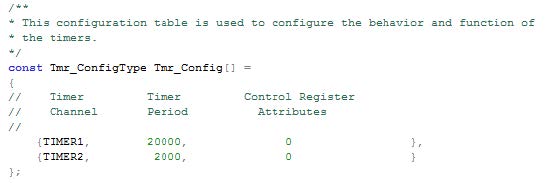
The Tmr\_ConfigType defines all of the data that is required in order to setup a single timer peripheral. Since most microcontrollers contain more than a single timer, an array of Tmr\_ConfigType would be created with each index of the array representing a channel (a single timer module). Before a configuration 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.

  
**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.

**Page 4**

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.

  
**Listing 8: Configuration Table Example for 2 timers**

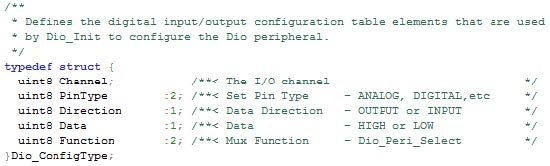
If Listing 8 was being used within a real project, the period values would correspond to the number of ticks of the timer required before an interrupt or some other useful system event would occur. The control register attributes would be representative of other registers that would require setup. It 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.

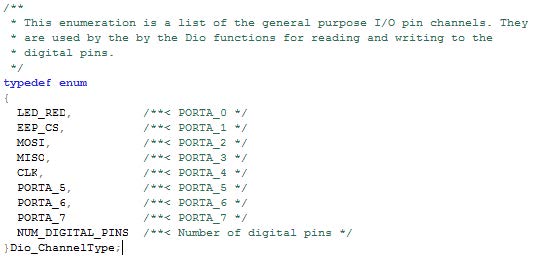
http://m.eet.com/media/1175350/jbdrv10.jpg  
**Listing 9: Example I/O Configuration**

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.

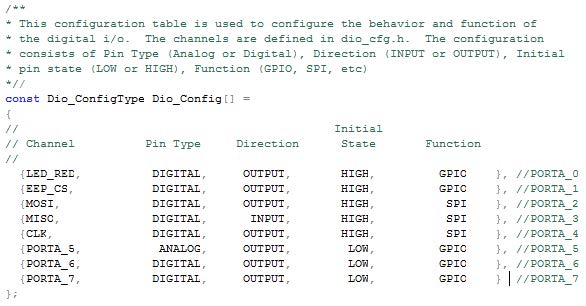
  
**Listing 10: Digital I/O Configuration Structure**

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.

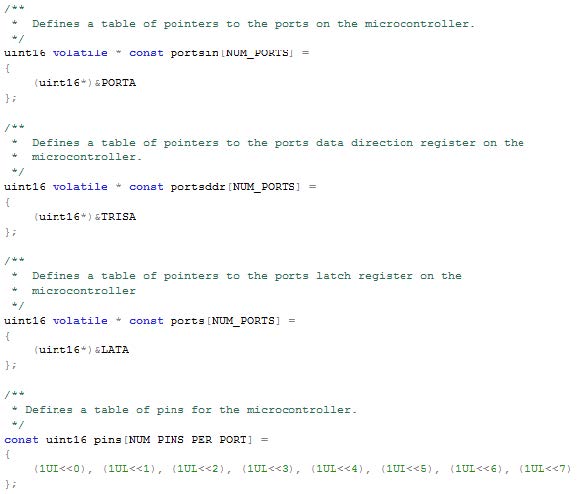
  
**Listing 11: Digital I/O Channel Types**

PAGE 5

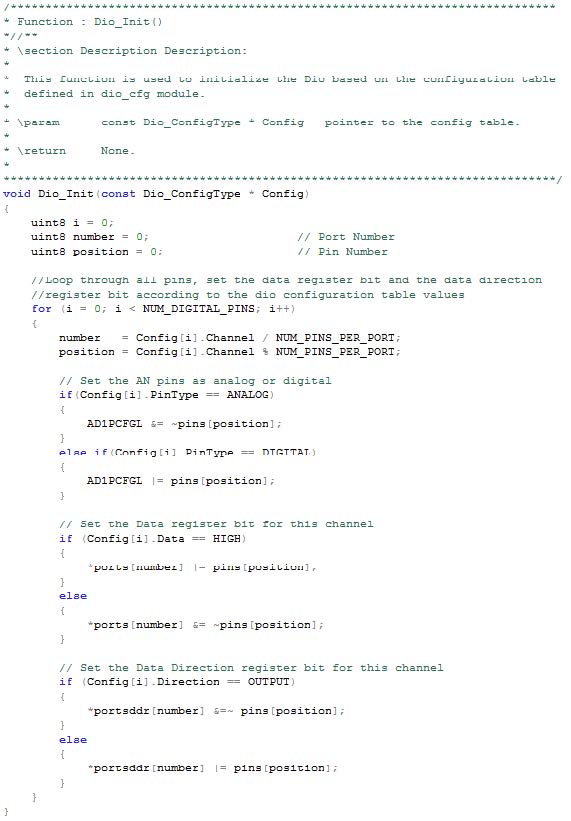
Once the channels have been defined it is straightforward to generate the configuration table. Create a const array that is of type Dio\_ConfigType and then start populating how each channel (pin) should be configured. For instance, for the LED\_RED channel, the pin should be configured as a digital 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.

  
**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.

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

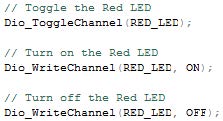
After a good amount of preparation the initialization function is finally ready to be written. It is relatively simple. A pointer to the configuration table is passed to the function. A simple loop is used to setup each of the pins. During each pass, each configuration value is read and based on the 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.

  
**Listing 14: Example Digital 1/0 Initialization Function**

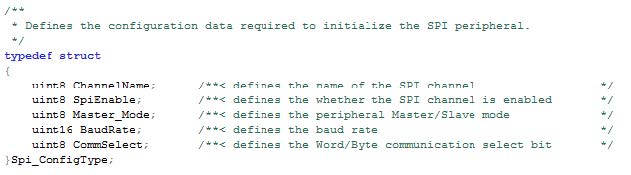
A quick example of how to write an additional function would be useful. In many applications it is often useful to toggle an led in order to see that the system is functioning. Listing 15 demonstrates how to access the pointer array to toggle a given channel.

  
**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.

  
**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.

  
**Listing 16: SPI Configuration Table Definitions**

***Jacob Beningo*** *is a Certified Software Development Professional (CSDP) who specializes in the development and design of quality, robust embedded systems. As a consultant, Jacob has worked in various industries including Automotive, Consumer, Defense, Medical and Space. He enjoys developing real-time software using the latest techniques in order to not only create code quickly and efficiently but also robustly. Jacob has been involved in numerous hardware development projects including navigational systems that have won the Best of CES award. His primary goal is to develop enabling technologies that revolutionize the way in which we do things. He has written technical papers on embedded design methods and taught courses on programmable device, boot-loaders and software methods at engineering conferences. He holds Bachelor’s degrees in Engineering and Physics from Central Michigan University and a Master’s degree in Space Systems Engineering from the University of Michigan.*

***This article was published previously on Embedded.com’s sister publication,*** [***EDN Magazine***](http://www.edn.com/design/systems-design/4402641/Developing-reusable-device-drivers-for-MCUs)