CYPRESS MICROSYSTEMS

Application Note **AN2135**

Calling Functions Using a Vector Table in C

By: Bill Keeler
Associated Project: Yes
Associated Part Family: CY8C25xxx, CY8C26xxx

Summary

This Application Note demonstrates how to use a vector table to call a function using indirection in C. This approach is very useful for creating powerful and easy-to-understand state machines.

Introduction

At times, we need a mechanism for executing code that is dependent on a state, or in response to an external command. There are two basic approaches: 1) Code a conditional statement using `if...else...if` constructs or a switch statement. 2) Create a vector table, and represent each state as an index into that table. Each approach has its advantages: The first option is attractive when there are just a few cases, code size and execution time is unimportant, and readability is not that significant. Option 2 shines when you have more than a few states, or stimuli, and when you want your code to execute quickly and be very readable.

Why Use Vectors?

Using a conditional statement requires that multiple expressions be written and evaluated in order to execute the desired code. If there are many states, these constructs can become very lengthy, and difficult to read. Using a vector table simply requires that the state has a known value, such as an enumeration. This state can be coded into the table and located by a search, or it may simply be an index into the table. Additionally, the vector table may be an array of structs, allowing you to provide more than just a pointer to the routine to be executed. You can, for example, supply a string that represents each state and also supply some state-specific data for each state.

How to Create a Vector Table

Because there's a separate memory partition for random-access and read-only data, the syntax for creating a vector table for the PSoC is slightly different than that for a conventional compiler for say, the PC. The compiler needs to know that we're putting this table in Flash, rather than in RAM. Therefore we have to declare our function pointer type slightly differently than we would for software that would run on a PC.

For example, writing code to execute on a PC we might declare:

```
typedef  void  (*psetter)(int);  //  declare
type psetter to be a function pointer
```

Then we'd create a table like this:

```
pseter PointerArray[2] = {
    Setter1,
    Setter2
};
```

We'd call that table like this:

```
 Setter[0](2);
 Setter[n](j);   // n and j are ints
```

…or, using a pointer, like this:

```
psetter p= Setter;
    *p(j);      // Call Setter1 with variable
j as the arg
    *(++p)(3)     //  call  setter2,  with  3  as
the arg
```

Because of its architecture, the PSoC calls for a slightly different approach:

```
typedef void const (*pSetter)(int);
//  use  the  const  specifier  to  tell  the
compiler that this points
// to an item in FLASH (not RAM)
```

We also have to store the pointer a bit differently, using an explicit cast:

```
const  psetter  PointerArray[2]  =  {  //  const
tells the compiler to put this in FLASH
    (pSetter) Setter1,
    (pSetter) Setter2
};
```

We call our function like this:

```
 (*(PointerArray[0]))(2);
 (*(PointerArray[n]))(j);     //  n  and  j  are
ints
```

…or, using a pointer, like this:

```
psetter p= Setter;
    *p(j);       // Call  Setter1  with  variable
j as the arg
    *(++p)(3)    //  call  setter2,  with  3  as
the arg
```

Or, we could have the vector be a member of a struct, and create an array of those structures. So, we'd declare it like this:

```
const struct {
    BYTE            DataByte;
    pCommandFunc    CmdFunc;
} VectorEntry[2] =
{ {1, (pSetter) Setter1}
  {5, (pSetter) Setter2} };
```

Then, we'd call it like this:

```
(*(VectorEntry[0].CmdFunc))(2);
// call Setter1
 (*(VectorEntry[StateNum].CmdFunc))(j);
// call Setter2, StateNum and j are ints
```

…or, using a pointer, like this:

```
psetter p = VectorEntry;
   (*p->CmdFunc)(j);
// Call Setter1 with variable j as the arg
    (*(++p->CmdFunc))(3);
// call setter2, with 3 as the arg, NOTE: p
now points to the second element
```

The Example Project

Now that we are able to use vector tables, let's discuss the example project. The project uses the PUP board, and merely demonstrates the concept of using vectors. It doesn't perform a particularly useful function.

Most of the processing takes place in the Interrupt Service Routine (ISR) for the Sleep Timer, which creates an interrupt once per-second. That ISR cycles the state counter.

In this example, the state serves as an index into the vector table. I included a PWM to flash one of the LEDs on the PUP board, but only to show the practicality of doing some common processing prior to calling the state vector routine. The state vector routines themselves only display different byte values on the bar-graph display. The purpose of that is to indicate which routine has executed most recently.

Ordinarily the state routine would be called via the vector table as a result of some event like a key press, a command or status received from a communications link, or a change in an I/O line.

Conclusion

Using vectors doesn't completely eliminate conditional logic associated with states in most cases. For example, processing often takes into account the previous, or current state prior to allowing a state change. However, using vector tables to encapsulate state routines and associated data, can save numerous lines of code, and make the logic much clearer. It is important to note the different way of declaring such pointers between the PSoC and a conventional computing platform. This is due to the Harvard architecture of the PSoC.

If you need to code complicated state machines, vectors are a nice addition to your toolbox.

About the Author

Name:	Bill Keeler
Title:	Systems Engineer
Background:	Fifteen Years of embedded systems design and programming. Mr. Keeler has designed systems for telecommunications, process instrumentation and control, Internet communications, and remote media control.
Contact:	bkPSoC@opctechnology.com