NIGEL JONES

Jump Tables via Function Pointer Arrays in C/C++

Jump tables, also called branch tables, are an efficient means of handling similar events in software. Here's a look at the use of arrays of function pointers in C/C++ as jump tables.

Examination of assembly language code that has been crafted by an expert will usually reveal extensive use of function "branch tables." Branch tables (a.k.a., jump tables) are used because they offer a unique blend of compactness and execution speed, particularly on microprocessors that support indexed addressing. When one examines typical C/C++ code, however, the branch table (i.e., an array of funtion pointers) is a much rarer beast. The purpose of this article is to examine why branch tables are not used by C/C++ programmers and to make the case for their extensive use. Real world examples of their use are included.

Function pointers

In talking to C/C++ programmers about this topic, three reasons are usually cited for not using function pointers. They are:

- They are dangerous
- A good optimizing compiler will generate a jump table from a switch statement, so let the compiler do the work
- They are too difficult to code and maintain

Are function pointers dangerous?

This school of thought comes about, because code that indexes into a table and then calls a function based on the index has the capability to end up just about anywhere. For instance, consider the following code fragment:

```
void (*pf[])(void) = {fna, fnb, fnc, …, fnz};
void test(const INT jump_index)
{
      /* Call the function specified by jump_index */
      pf[jump_index]();
}
```

The above code declares `pf[]` to be an array of pointers to functions, each of which takes no arguments and returns void. The `test()` function simply calls the specified function via the array. As it stands, this code is dangerous for the following reasons.

- `pf[]` is accessible by anyone
- In `test()`, there is no bounds checking, such that an erroneous `jump_index` would spell disaster

A much better way to code this that avoids these problems is as follows

```
void test(uint8_t const ump_index)
{
      static void (*pf[])(void) = {fna, fnb, fnc, …, fnz};
      if (jump_index < sizeof(pf) / sizeof(*pf))
      {
            /* Call the function specified by jump_index */
            pf[jump_index]();
      }
}
```

The changes are subtle, yet important.

- By declaring the array static within the function, no one else can access the jump table
- Forcing `jump_index` to be an unsigned quantity means that we only need to perform a one sided test for our bounds checking
- Setting `jump_index` to the smallest data type possible that will meet the requirements provides a little more protection (most jump tables are smaller than 256 entries)
- An explicit test is performed prior to making the call, thus ensuring that only valid function calls are made. (For performance critical applications, the `if()` statement could be replaced by an `assert()`)

This approach to the use of a jump table is just as secure as an explicit switch statement, thus the idea that jump tables are dangerous may be rejected.

Leave it to the optimizer?

It is well known that many C compilers will attempt to convert a switch statement into a jump table. Thus, rather than use a function pointer array, many programmers prefer to use a switch statement of the form:

```
void test(uint8_t j const ump_index)
{
      switch (jump_index)
      {
      case 0:
            fna();
            break;
      case 1: fnb();
```

```
                        break;
                        …
                case 26:
                        fnz();
                        break;
                default:
                        break;
                }
        }
```

Indeed, Jack Crenshaw advocated this approach in a September 1998 column in *Embedded Systems Programming*. Well, I have never found myself disagreeing with Dr. Crenshaw before, but there is always a first time for everything! A quick survey of the documentation for a number of compilers revealed some interesting variations. They all claimed to potentially perform conversion of a switch statement into a jump table. However, the criteria for doing so varied considerably. One vendor simply said that they would attempt to perform this optimization. A second claimed to use a heuristic algorithm to decide which was "better," while a third permitted pragma's to let the user specify what they wanted. This sort of variation does not give one a warm fuzzy feeling!

In the case where one has, say, 26 contiguous indices, each associated with a single function call (such as the example above), the compiler will almost certainly generate a jump table. However, what about the case where you have 26 non-contiguous indices, that vary in value from 0 to 1000? A jump table would have 974 null entries or 1948 "wasted" bytes on the average microcontroller. Most compilers would deem this too high a penalty to pay, and would eschew the jump table for an if-else sequence. However, if you have EPROM to burn, it actually costs nothing to implement this as a jump table, but buys you consistent (and fast) execution time. By coding this as a jump table, you ensure that the compiler does what you want.

There is a further problem with large switch statements. Once a switch statement gets much beyond a screen length, it becomes harder to see the big picture, and thus the code is more difficult to maintain. A function pointer array declaration, adequately commented to explain the declaration, is much more compact, allowing one to see the overall picture. Furthermore, the function pointer array is potentially more robust. Who has not written a large switch statement and forgotten to add a break statement on one of the cases?

Complexities

Complexity associated with jump table declaration and use is the real reason they are not used more often. In embedded systems, where pointers normally have mandatory memory space qualifiers, the declarations can quickly become horrific. For instance, the example above would be highly undesirable on most embedded systems, since the pf[] array would probably end up being stored in RAM, instead of ROM. The way to ensure the array is stored in ROM varies somewhat between compiler vendors. However, a first step that is portable to all systems is to add const qualifiers to the declaration. Thus, our array declaration now becomes:

```
static void (* const pf[])(void) = {fna, fnb, fnc, …, fnz};
```

Like many users, I find these declarations cryptic and very daunting. However, over the years, I have built up a library of declaration templates that I simply refer to as necessary. A compilation of useful templates appears below.

A handy trick is to learn to read complex declarations like this backwards–i.e., from right to left. Doing this here's how I'd read the above: `pf` is an array of constant pointers to functions that return void. The static keyword is only needed if this is declared privately within the function that uses it–and thus keeping it off the stack.

Arrays of function pointers

Most books about C programming cover function pointers in less than a page (while devoting entire chapters to simple looping constructs). The descriptions typically say something to the effect that you can take the address of a function, and thus one

can define a pointer to a function, and the syntax looks like such and such. At which point, most readers are left staring at a complex declaration, and wondering what exactly function pointers are good for. Small wonder that function pointers do not feature heavily in their work.

Well then, where are jump tables useful? In general, arrays of function pointers are useful whenever there is the potential for a range of inputs to a program that subsequently alters program flow. Some typical examples from the embedded world are given below.

Keypads

The most often cited example for uses of function pointers is with keypads. The general idea is obvious. A keypad is normally arranged to produce a unique keycode. Based on the value of the key pressed, some action is taken. This can be handled via a switch statement. However, an array of function pointers is far more elegant. This is particularly true when the application has multiple user screens, with some key definitions changing from screen to screen (i.e., the system uses soft keys). In this case, a two dimensional array of function pointers is often used.

```
#define N_SCREENS  16
#define N_KEYS     6
/* Prototypes for functions that appear in the jump table */
INT fnUp(void);
INT fnDown(void);
…
INT fnMenu(void);
INT fnNull(void);
INT keypress(uint8_t key, uint8_t screen)
{
      static INT (* const pf[N_SCREENS][N_KEYS])(void) =
      {
```

```
                  {fnUp, fnDown, fnNull, …, fnMenu},
                  {fnMenu, fnNull, …, fnHome},
                  …
                  {fnF0, fnF1, …, fnF5}
            };
            assert (key < N_KEYS); assert( screen < N_SCREENS);
            /* Call the function and return result */
            return (*pf[screen][key])();
      }
      /* Dummy function - used as an array filler */
      INT fnNull(void)
      {
            return 0;
      }
```

There are several points to note about the above example:

- All functions to be named in a jump table should be prototyped. Prototyping is the best line of defense against including a function that expects the wrong parameters, or returns the wrong type.
- As for earlier examples, the function table is declared within the function that makes use of it (and, thus, static)
- The array is made const signifying that we wish it to remain unchanged
- The indices into the array are unsigned, such that only single sided bounds checking need be done
- In this case, I have chosen to use the `assert()` macro to provide the bounds checking. This is a good compromise between debugging ease and runtime efficiency.
- A dummy function `fnNull()` has been declared. This is used where a keypress is undefined. Rather than explicitly testing to see whether a key is valid, the dummy function is invoked. This is usually the most efficient method of handling an function array that is only partially populated.
- The functions that are called need not be unique. For example, a function such as `fnMenu` may appear many times in the same jump table.

Communication protocols

Although the keypad example is easy to appreciate, my experience in embedded systems is that communication links occur far more often than keypads. Communication protocols are a challenge ripe for a branch table solution. This is best illustrated by an example.

Last year, I worked on the design for an interface box to a very large industrial power supply. This interface box had to accept commands and return parameter values over a RS-232 link. The communications used a set of simple ASCII mnemonics to specify the action to be taken. The mnemonics consisted of a channel number (0,1, or 2), followed by a two character parameter. The code to handle a read request coming in over the serial link is shown below. The function `process_read()` is called with a pointer to a string fragment that is expected to consist of the three characters (null terminated) containing the required command.

```
const CHAR *fna(void);  // Example function prototype
static void process_read(const CHAR *buf)
{
      CHAR *cmdptr;
      UCHAR offset;
      const CHAR *replyptr;

      static const CHAR read_str[] =
       "0SV 0SN 0MO 0WF 0MT 0MP 0SW 1SP 1VO 1CC 1CA 1CB
        1ST 1MF 1CL 1SZ 1SS 1AZ 1AS 1BZ 1BS 1VZ 1VS 1MZ
        1MS 2SP 2VO 2CC 2CA 2CB 2ST 2MF 2CL 2SZ 2SS
        2AZ 2AS 2BZ 2BS 2VZ 2VS 2MZ 2MS ";
static const CHAR *
      (* const readfns[sizeof(read_str)/4])(void) =
      {
            fna,fnb,fnc, …
      };
cmdptr = strstr(read_str, buf);

if (cmdptr != NULL)
      {
            /*
            * cmdptr points to the valid command, so compute offset,
            * in order to get entry into function jump table
            */
            offset = (cmdptr - read_str) / 4;
            /* Call function and get pointer to reply*/
            replyptr = (*readfns[offset])();
            /* rest of the code goes here */
      }
}
```

The code above is quite straightforward. A constant string, `read_str`, is defined. The `read_str` contains the list of all legal mnemonic combinations. Note the use of added spaces to aid clarity. Next, we have the array of function pointers, one pointer for each valid command. We determine if we have a valid command sequence by making use of the standard library function `strstr()`. If a match is found, it returns a pointer to the matching substring, else it returns NULL. We check for a valid pointer, compute the offset into the string, and then use the offset to call the appropriate handler function in the jump table. Thus, in four lines of code, we have determined if the command is valid and called the appropriate function. Although the declaration of `readfns[]` is complex, the simplicity of the runtime code is tough to beat.

Timed task list

A third area where function pointers are useful is in timed task lists. In this case, the input to the system is the passage of time. Many projects cannot justify the use of an RTOS. Instead, all that is required is that a number of tasks run at predetermined intervals. This is very simply handled as shown below.

```
typedef struct
{
      UCHAR interval;   /* How often to call the task */
      void (*proc)(void);     /* pointer to function returning void */
} TIMED_TASK;

static const TIMED_TASK timed_task[] =
{
      { INTERVAL_16_MSEC,     fnA },
      { INTERVAL_50_MSEC,     fnB },
      { INTERVAL_500_MSEC, fnC },
      …
      { 0, NULL }
};

extern volatile UCHAR tick;

void main(void)
{
      const TIMED_TASK *ptr;
      UCHAR time;
      /* Initialization code goes here. Then enter the main loop */
      while (1)
      {
if (tick)
            {
                  /* Check timed task list */
                  tick--;
                  time = computeElapsedTime(tick);
                  for (ptr = timed_task; ptr->interval !=0; ptr++)
                  {
                        if (!(time % ptr->interval))
                        {
                              /* Time to call the function */
                  (ptr->proc)();
                        }
                  }
            }
      }
}
```

In this case, we define our own data type (TIMED_TASK) that consists simply of an interval and a pointer to a function. We then define an array of TIMED_TASK, and initialize it with the list of functions that are to be called and their calling interval. In `main()`, we have the start up code which must enable a periodic timer interrupt that increments the volatile variable tick at a fixed interval. We then enter the infinite loop.

The infinite loop checks for a non-zero tick value, decrements the tick variable and computes the elapsed time since the program started running. The code then simply steps through each of the tasks, to see whether it is time for that one to be executed and, if so, calls it via the function pointer.

If your application only consists of two or three tasks, then this approach is probably overkill. However, if your project has a large number of timed tasks, or it is likely that you will have to add tasks in the future, then this approach is rather palatable. Note that adding tasks and/or changing intervals simply requires editing of the timed_task[] array. No code, per se, has to be changed.

Interrupt vector tables

The fourth application of function jump tables is the array of interrupt vectors. On most processors, the interrupt vectors are in contiguous locations, with each vector representing a pointer to an interrupt service routine function. Depending upon the compiler, the work may be done for you implicitly, or you may be forced to generate the function table. In the latter case, implementing the vectors via a switch statement will not work!

Here is the vector table from the industrial power supply project mentioned above. This project was implemented using a Whitesmiths' compiler and a 68HC11 microncontroller.

```
IMPORT VOID _stext();   /* 68HC11-specific startup routine */
static VOID (* const _vectab[])() =
{
      SCI_Interrupt,    /* SCI                    */
      badSPI_Interrupt, /* SPI                    */
      badPAI_Interrupt, /* Pulse acc input        */
      badPAO_Interrupt, /* Pulse acc overf        */
      badTO_Interrupt,  /* Timer overf            */
      badOC5_Interrupt, /* Output compare 5       */
      badOC4_Interrupt, /* Output compare 4       */
      badOC3_Interrupt, /* Output compare 3       */
      badOC2_Interrupt, /* Output compare 2       */
      badOC1_Interrupt, /* Output compare 1       */
      badIC3_Interrupt, /* Input capture 3        */
      badIC2_Interrupt, /* Input capture 2        */
      badIC1_Interrupt, /* Input capture 1        */
      RTI_Interrupt,    /* Real time              */
      Uart_Interrupt,   /* IRQ                    */
      PFI_Interrupt,    /* XIRQ                   */
      badSWI_Interrupt, /* SWI                    */
      IlOpC_Interrupt,  /* illegal                */
      _stext,     /* cop fail                     */
      _stext,     /* cop clock fail               */
      _stext,     /* RESET                        */
};
```

A couple of points are worth making:

- The above is insufficient to locate the table correctly in memory. This has to be done via linker directives.
- Note that unused interrupts still have an entry in the table. Doing so ensures that the table is correctly aligned and traps can be placed on unexpected interrupts.

If any of these examples has whet your appetite for using arrays of function pointers, but you are still uncomfortable with the declaration complexity, then fear not! You will find a variety of declarations, ranging from the straightforward to the downright appalling below. The examples are all reasonably practical in the sense that the desired functionality is not outlandish (that is, there are no declarations for arrays of pointers to functions that take pointers to arrays of function pointers and so on).

Declaration and use hints

All of the examples below adhere to conventions that I have found to be useful over the years, specifically:

1. *All of the examples are preceded by static. This is done on the assumption that the scope of a function table should be highly localized, ideally within an enclosing function.*

2. *In every example the array* `pf[]` *is also preceded with const. This declares that the pointers in the array cannot be modified after initialization. This is the normal (and safe) usage scenario.*

3. *There are two syntactically different ways of invoking a function via a pointer. If we have a function pointer with the declaration:*

   ```
   void (*fnptr)(int);  /* fnptr is a function pointer */
   ```

   Then it may be invoked using either of these methods:

   ```
   fnptr(3);            /* Method 1 of invoking the function */
   (*fnptr)(3);         /* Method 2 of invoking the function */
   ```

   The advantage of the first method is an uncluttered syntax. However, it makes it look as if fnptr is a function, as opposed to being a function pointer. Someone maintaining the code may end up searching in vain for the function fnptr(). With method 2, it is much clearer that we are dereferencing a pointer. However, when the declarations get complex, the added (*) can be a significant burden. Throughout the examples, each syntax is shown. In practice, the latter syntax seems to be more popular–and you should use only one.

4. *In every example, the syntax for using a typedef is also given. It is quite permissible to use a typedef to define a complex declaration, and then use the new type like a simple type. If we stay with the example above, then an alternative declaration is:*

   ```
   typedef void (*PFV_I )(int);

   /* Declare a PVFV_I typed variable and init it */
   PFV_I fnptr = fna;

   /* Call fna with parameter 3 using method 1 */
   fnptr(3);

   /* Call fna with parameter 3 using method 2 */
   (*fnptr)(3);
   ```

The `typedef` declares the type PFV_I to be a pointer to a function that returns void and is passed an integer. We then simply declare fnptr to a variable of this type, and use it. `typedefs` are very good when you regularly use a certain function pointer type, since it saves you having to remember and type in the declaration. The downside of using a `typedef`, is the fact that it is not obvious that the variable that has been declared is a pointer to a function. Thus, just as for the two invocation methods above, you can gain syntactical simplicity by hiding the underlying functionality.

In the `typedefs`, a consistent naming convention is used. Every type starts with PF (Pointer to Function) and is then followed with the return type, followed by an underscore, the first parameter type, underscore, second parameter type and so on. For void, boolean, char, int, long, float and double, the characters V, B, C, I, L, S, D are used. (Note the use of S(ingle) for float, to avoid confusion with F(unction)). For a pointer to a data type, the type is preceded with P. Thus PL is a pointer to a long. If a parameter is const, then a c appears in the appropriate place. Thus, cPL is a const pointer to a long, whereas a PcL is a pointer to a const long, and cPcL is a const pointer to a const long. For volatile qualifiers, v is used. For unsigned types, a u precedes the base type. For user defined data types, you are on your own!

An extreme example: PFcPcI_uI_PvuC. This is a pointer to a function that returns a `const` pointer to a `const` Integer that is passed an unsigned integer and a pointer to a volatile unsigned char.

Function pointer templates

The first eleven examples are generic in the sense that they do not use memory space qualifiers and hence may be used on any target. Example 12 shows how to add memory space qualifiers, such that all the components of the declaration end up in the correct memory spaces.

Example 1

`pf[]` is a static array of pointers to functions that take an INT as an argument and return void.

```
void fna(INT); // Example prototype of a function to be called

// Declaration using typedef typedef void (* const PFV_I)(INT);
static PFV_I pf[] = {fna,fnb,fnc, … fnz);

// Direct declaration
static void (* const pf[])(INT) = {fna, fnb, fnc, … fnz};

// Example use
INT a = 6;
pf[jump_index](a);             // Calling method 1
(*pf[jump_index])(a);   // Calling method 2
```

Example 2

`pf[]` is a static array of pointers to functions that take a pointer to an INT as an argument and return void.

```
void fna(INT *);  // Example prototype of a function to be called

// Declaration using typedef
typedef void (* const PFV_PI)(INT *);
static PVF_PI[] = {fna,fnb,fnc, … fnz};

// Direct declaration
static void (* const pf[])(INT *) = {fna, fnb, fnc, … fnz};

// Example use
INT a = 6;
pf[jump_index](&a);     // Calling method 1
(*pf[jump_index])(&a);  // Calling method 2
```

Example 3

`pf[]` is a static array of pointers to functions that take an INT as an argument and return a CHAR.

```
CHAR fna(INT);    // Example prototype of a function to be called

// Declaration using typedef
typedef CHAR (* const PFC_I)(INT);
static PVC_I[] = {fna,fnb,fnc, … fnz};

// Direct declaration
static CHAR (* const pf[])(INT) = {fna, fnb, fnc, … fnz};

// Example use
INT a = 6;
CHAR res;
res = pf[jump_index](a);            // Calling method 1
res = (*pf[jump_index])(a);   // Calling method 2
```

Example 4

`pf[]` is a static array of pointers to functions that take an INT as an argument and return a pointer to a CHAR.

```
CHAR *fna(INT);   // Example prototype of a function to be called

// Declaration using typedef
typedef CHAR * (* const PFPC_I)(INT);
static PVPC_I[] = {fna,fnb,fnc, … fnz};

// Direct declaration
static CHAR * (* const pf[])(INT) = {fna, fnb, fnc, … fnz};

// Example use
INT a = 6;
```

```
        CHAR * res;
        res = pf[jump_index](a);      // Calling method 1
        res = (*pf[jump_index])(a); // Calling method 2
```

Example 5

`pf[]` is a static array of pointers to functions that take an INT as an argument and return a pointer to a const CHAR (i.e. the pointer may be modified, but what it points to may not).

```
        const CHAR *fna(INT);   // Example prototype of a function to be called

        // Declaration using typedef
        typedef const CHAR * (* const PFPcC_I)(INT);
        static PVPcC_I[] = {fna,fnb,fnc, … fnz};

        // Direct declaration
        static const CHAR * (* const pf[])(INT) = {fna, fnb, fnc, … fnz};

        // Example use INT a = 6;
        const CHAR * res;
        res = pf[jump_index](a);            //Calling method 1
        res = (*pf[jump_index])(a);   //Calling method 2
```

Example 6

`pf[]` is a static array of pointers to functions that take an INT as an argument and return a const pointer to a CHAR (i.e. the pointer may not be modified, but what it points to may be modified).

```
        CHAR * const fna(INT i);      // Example prototype of a function to be called

        // Declaration using typedef
        typedef CHAR * const (* const PFcPC_I)(INT);
        static PVcPC_I[] = {fna,fnb,fnc, … fnz};

        // Direct declaration
        static CHAR * const (* const pf[])(INT) = {fna, fnb, fnc, … fnz};

        // Example use
        INT a = 6;
        CHAR * const res = pf[jump_index](a);     // Calling method 1
        CHAR * const res = (*pf[jump_index])(a);  // Calling method 2
```

Example 7

`pf[]` is a static array of pointers to functions that take a pointer to a const INT as an argument (i.e. the pointer may be modified, but what it points to may not) and return a const pointer to a const CHAR (i.e. the pointer, nor what it points to may be modified).

```
        const CHAR * const fna(const INT *i);     // Example prototype

        // Declaration using typedef
        typedef const CHAR * const (* const PFcPcC_PcI)(const INT *);
```

```
static PVcPcC_PcI[] = {fna,fnb,fnc, … fnz};

// Direct declaration
static const CHAR * const (* const pf[])(const INT *) = {fna, fnb, fnc, … fnz};

// Example use
INT a = 6;
const CHAR* const res = pf[jump_index](aptr);   //Calling method 1
const CHAR* const res = (*pf[jump_index])(aptr);//Calling method 2
```

Example 8

`pf[]` is a static array of pointers to functions that take a const pointer to an INT as an argument (i.e. the pointer may not be modified, but what it points to may) and return a const pointer to a const CHAR (i.e. the pointer, nor what it points to may be modified)

```
const CHAR * const fna(INT *const i);     // Example prototype

// Declaration using typedef
typedef const CHAR * const (* const PFcPcC_cPI)(INT * const);
static PVcPcC_cPI[] = {fna,fnb,fnc, … fnz};

// Direct declaration
static const CHAR * const (* const pf[])(INT * const) = {fna, fnb, fnc, … fnz};

// Example use
const INT a = 6;
const INT *aptr;
aptr = &a;
const CHAR* const res = pf[jump_index](aptr);   //Calling method 1
const CHAR* const res = (*pf[jump_index])(aptr);//Calling method 2
```

Example 9

`pf[]` is a static array of pointers to functions that take a const pointer to a const INT as an argument (i.e. the pointer nor what it points to may be modified) and return a const pointer to a const CHAR (i.e. the pointer, nor what it points to may be modified)

```
const CHAR * const fna(const INT *const i);     // Example prototype

// Declaration using typedef
typedef const CHAR * const (* const PFcPcC_cPcI)(const INT * const);
static PVcPcC_cPcI[] = {fna,fnb,fnc, … fnz};

// Direct declaration
static const CHAR * const (* const pf[])(const INT * const) = {fna, fnb, fnc, …
fnz};

// Example use
INT a = 6;
INT *const aptr = &a;
const CHAR* const res = pf[jump_index](aptr);         // Method 1
const CHAR* const res = (*pf[jump_index])(aptr);      // Method 2
```

Example 10

`pf[]` is a static array of pointers to functions that take a const pointer to a const INT as an argument (i.e. the pointer nor what it points to may be modified) and return a const pointer to a volatile CHAR (i.e. the pointer may not be modified, but what it points to may change unexpectedly)

```
const CHAR * fna(const INT *const i);     // Example prototype

// Declaration using typdef
typedef const CHAR * const (* const PFcPcC_cPcI) (const INT * const);
static PVcPcC_cPcI[] = {fna, fnb, fnc, … fnz};

// Direct declaration
static const CHAR * const (* const pf[]) (const INT * const) = (fna, fnb, fnc, …
fnz};

// Example use
const INT a = 6;
const INT *const aptr = &a;

const CHAR* const res = pf[jump_index](aptr);          // Method 1
const CHAR* const res = (*pf[jump_index])(aptr);       // Method 2
```

This example manages to combine five incidences of const and one of static into a single declaration. For all of its complexity, however, this is not an artificial example. You could go ahead and remove all the const and static declarations and the code would still work. It would, however, be a lot less safe, and potentially less efficient.

Just to break up the monotony, here is the same declaration, but with a twist.

Example 11

`pf[]` is a static array of pointers to functions that take a const pointer to a const INT as an argument (i.e. the pointer nor what it points to may be modified) and return a const pointer to a volatile CHAR (i.e. the pointer may not be modified, but what it points to may change unexpectedly)

```
volatile CHAR * const fna(const INT *const i);  // Example prototype

// Declaration using typedef
typedef volatile CHAR * const (* const PFcPvC_cPcI)(const INT * const); static
PVcPvC_cPcI[] = {fna,fnb,fnc, … fnz};

// Direct declaration
static volatile CHAR * const (* const pf[])(const INT * const) = {fna, fnb, fnc, …
fnz};

// Example use
const INT a = 6;
const INT * const aptr = &a;

volatile CHAR * const res = pf[jump_index](aptr);     // Method 1
volatile CHAR * const res = (*pf[jump_index])(aptr);  // Method 2
```

```
while (*res)
;     //Wait for volatile register to clear
```

With memory space qualifiers, things can get even more hairy. For most vendors, the memory space qualifier is treated syntactically as a type qualifier (such as const or volatile) and thus follows the same placement rules. For consistency, I place type qualifiers to the left of the “thing” being qualified. Where there are multiple type qualifiers, alphabetic ordering is used. Since memory space qualifiers are typically compiler extensions, they are normally preceded by an underscore, and hence come first alphabetically. Thus, a nasty declaration may look like this:

```
_ram const volatile UCHAR status_register;
```

To demonstrate memory space qualifier use, here is example 11 again, except this time memory space qualifiers have been added. The qualifiers are named _m1 … _m5.

Example 12

`pf[]` is a static array of pointers to functions that take a const pointer to a const INT as an argument (i.e. the pointer nor what it points to may be modified) and return a const pointer to a volatile CHAR (i.e. the pointer may be modified, but what it points to may change unexpectedly). Each element of the declaration lies in a different memory space. In this particular case, it is assumed that you can even declare the memory space in which parameters passed by value appear. This is extreme, but is justified on pedagogical grounds.

```
/* An example prototype. This declaration reads as follows.
 * Function fna is passed a const pointer in _m5 space that points to a
 * const integer in _m4 space. It returns a const pointer in _m2 space to
 * a volatile character in _m1 space.
 */
_m1 volatile CHAR * _m2 const fna(_m4 const INT * _m5 const i);

/* Declaration using typedef. This declaration reads as follows.
 * PFcPvC_cPcI is a pointer to function data type, variables based
 * upon which lie in _m3 space. Each Function is passed a const
 * pointer in _m5 space that points to a const integer in _m4 space.
 * It returns a const pointer in _m2 space to a volatile character
 * in _m1 space.
*/
typedef _m1 volatile CHAR * _m2 const (* _m3 const PFcPvC_cPcI) (_m4 const INT * _m5
const);
static PVcPvC_cPcI[] = {fna,fnb,fnc, … fnz};

/* Direct declaration. This declaration reads as follows. pf[] is
 * a statically allocated constant array in _m3 space of pointers to functions.
 * Each Function is passed a const pointer in _m5 space that points to
 * a const integer in _m4 space. It returns a const pointer in _m2 space
 * to a volatile character in _m1 space.
 */
static _m1 volatile CHAR * _m2 const (* _m3 const pf[]) (_m4 const INT * _m5 const)
= {fna, fnb, fnc, … fnz};

// Declare a const variable that lies in _m4 space
```

```
_m4 const INT a = 6;

// Now declare a const pointer in _m5 space that points to a const
// variable that is in _m4 space
_m4 const INT * _m5 const aptr = &a;

// Make the function call, and get back the pointer
volatile CHAR * const   res = pf[jump_index](&a);     //Method 1
volatile CHAR * const   res = (*pf[jump_index])(&a);  //Method 2

while (*res)
;     // Wait for volatile register to clear
```

Acknowledgments

My thanks to Mike Stevens not only for reading over this manuscript and making some excellent suggestions but also for over the years showing me more ways to use function pointers that I ever dreamed was possible.

This article was published in the May 1999 issue of *Embedded Systems Programming*. If you wish to cite the article in your own work, you may find the following MLA-style information helpful:

Jones, Nigel. "Arrays of Pointers to Functions" *Embedded Systems Programming*, May 1999.