

Arrays of Pointers to Functions

In typical C code, the function jump table is not widely used. Here the author examines why jump tables aren't exploited more frequently and makes a case for their increased usage with real-world examples.

When you examine assembly language code that has been crafted by an expert, you'll usually find extensive use of function jump tables. These jump tables, also known as an array of pointers to functions, are used because they offer a unique blend of compactness and execution speed, particularly on microprocessors that support indexed addressing.

When you examine typical C code, however, the function jump table is a much rarer beast. In this article, I will examine why jump tables aren't used more often, and make the case for their extensive use. I've also included real-world examples of their use.

Why aren't function pointers popular?

Developers usually cite three reasons for not using function pointers. These reasons are:

- Function pointers are dangerous
- Good optimizing compilers generate a jump table from

a switch statement, so letting the compiler do the work is preferable

- Function pointers are too difficult to code and maintain

The perceived danger of function pointers

The "function pointers are dangerous" school of thought comes about because code that indexes into a table and then calls a function based on the index can 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]();
}
```

This fragment declares pf to be an array of pointers to func-

The function pointer array is potentially more robust. Who hasn't written a large switch statement and forgotten to add a break statement on one of the cases?

tions that take no arguments and return void. The test function simply calls the specified function. As it stands, this code is dangerous for the following reasons:

- `pf[]` can be accessed by anyone
- No bounds checking is present in `test()`, such that an erroneous `jump_index` would spell disaster

Consider this alternative approach, which avoids these problems:

```
void test(const UCHAR jump_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, but important. We accomplish a number of things with this approach:

- 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, thereby making a good argument against the perceived danger of function pointers.

Leave it to the optimizer

Many compilers will attempt to convert a switch statement into a jump table. Thus, rather than use a function pointer array, many users prefer to use a switch statement of the form:

```
void test(const UCHAR jump_index)
{
    switch jump_index
    {
        case 0:
            fa();
            break;
        case 1:
            fb();
            break;
        ...
        case 26:
            fz();
            break;
        default:
            break;
    }
}
```

Indeed, *ESP* columnist Jack Crenshaw advocates this approach (“Interpreters—A Recap,” September 1998, p. 23). Though I’ve never disagreed with Dr. Crenshaw before, there’s 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. But the criteria for doing so varied considerably. One vendor simply said their compiler would attempt to perform this optimization. A second claimed to use a heuristic algorithm to decide

which was “better,” while a third permitted pragmas to let users specify what they wanted. This degree of variation doesn’t give me a warm fuzzy feeling.

In the case where you have, say, 26 contiguous indices, each associated with a single function call (such as the previous example), then you can almost guarantee that the compiler will generate a jump table. But what about the case in which you have 26 non-contiguous indices that vary in value from zero to 1,000? A jump table would have 974 null entries, or 1,948 “wasted” bytes on a typical microcontroller. Most compilers would deem this too high a penalty to pay, and would eschew the jump table for an if-then-else-if sequence. However, if you have EPROM to burn, implementing this as a jump table actually costs nothing and buys you consistent (and fast) execution time. By coding this as a jump table, you ensure that the compiler does what you want.

Large switch statements present another problem. Once a switch statement gets too far beyond a screen length, seeing the big picture becomes harder, making the code more difficult to maintain. A function pointer array declaration, adequately commented to explain the declaration, is far more compact and allows you to see the overall picture. Furthermore, the function pointer array is potentially more robust. Who hasn’t written a large switch statement and forgotten to add a break statement on one of the cases?

Declaration and use complexity

Declaration and use complexity are the real reasons that jump tables aren’t used more often. In embedded systems, where pointers normally have

mandatory memory space qualifiers, declarations can quickly become horrific. For instance, the previous example would be highly undesirable on most embedded systems because 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 among compilers. However, a first step, which is portable to all systems, is to add `const` qualifiers to the declaration. Thus, our array declaration becomes:

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

Like many users, I find these declarations cryptic and daunting. But over the years, I've built up a library of declaration templates to which I simply refer when necessary. You'll find this list of templates on the *ESP* Web site at www.embedded.com/code.htm.

Applications of function pointers

Most textbooks cover function pointers in less than one page (while devoting entire chapters to simple looping constructs). The descriptions typically say that you can take the address of a function—and therefore define a pointer to a function—and the syntax looks like so-and-so. At this point, most users are left with a complex declaration and are wondering exactly what function pointers are good for. Small wonder, then, that function pointers do not figure heavily in their work.

Well then, when are jump tables useful? Arrays of function pointers are generally useful whenever the potential exists for a variety of input into the program that alters the program flow. Some typical examples from the embedded world follow.

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

Most textbooks cover function pointers in less than one page (while devoting entire chapters to simple looping constructs).

is taken. As usual, this action can be handled via a switch statement. However, an array of function pointers can be far more elegant, particularly

when the application has multiple screens, and the key definition changes from screen to screen (i.e., the system uses soft keys). In this case,

LISTING 1 A two-dimensional array of function pointers is often used in keypad applications.

```

#define N_SCREEN      16
#define N_KEYS        6
/* Function prototypes for functions that appear in the jump table */
INT fn1(void);
INT fn2(void);
...
INT fn60(void);
INT fnNull(void);

INT keypress(UCHAR key, UCHAR screen)
{
    static INT (* const pf[N_SCREEN][N_KEYS])(void) = {
        {fn1, fn2, fnNull, fn8},
        {fn9, fnNull, fn12, fn7},
        ...
        {fn50, fn51, fn60} };

    assert (key < N_KEYS);
    assert (screen < N_SCREEN);
    return (*pf[screen][key])();    //Call the function and return result
}

INT fnNull(void)
{
    /* Dummy function used as an array filler */
    return 0;
}

```

a two-dimensional array of function pointers is often used. See Listing 1 for an example.

Note several points about this example:

- All functions to be named in a function table should be prototyped. This precaution is your 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 static in the function that makes use of it
- The array is made const, signifying that we wish it to remain unchanged
- The indices into the array are unsigned, so that only single-sided bounds checking needs to be done
- In this case, I've chosen to use the

assert() macro to provide the bounds checking. This approach is an excellent compromise between ease of debugging and run-time efficiency

- A dummy function, fnNull(), has been declared. This function is used where a keypress is undefined. Rather than explicitly testing to see whether a key is valid, you can simply call a dummy function, which is usually the most efficient method of handling an array that is only partially populated
- The functions that are called need not be unique. Rather, a function can appear many times in the same array

Communication links. Although the keypad example is easy to appreciate, my experience in embedded systems has revealed that communication links

LISTING 2 Code to handle a read request coming in over the serial link

```
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[] =
    "OSV OSN OMO OWF OMT OMP OSW 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 2VS 2VZ 2VS
    2MZ 2MS";

    static const CHAR * (*const readfns[sizeof(read_str)/4])(void) = {
        fna,fnb,fnb, ... };

    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 + 1 - read_str) / 4;

        /* Call function, & get pointer to reply*/
        replyptr = (*readfns[offset])();
        /* rest of the code goes here */
    }
}
```

Although the declaration of `readfns[]` is complex, the simplicity of the run-time code is difficult to beat.

occur far more often than keypads. Communication links are applications ripe for function tables.

Last year, I worked on the design for an interface box to a large industrial power supply. This interface box had to accept commands and return parameter values over an RS-232 link. The communication used a set of simple ASCII mnemonics to specify the action to be taken. The mnemonics consisted of a channel number (zero, one, or two), followed by a two-character parameter. The code to handle a

read request coming in over the serial link is shown in Listing 2. The function `process_read()` is called with a pointer to a string fragment which should consist of the three characters (null-terminated) containing the required command.

The code shown in Listing 2 is quite simple. We define a constant string, `read_str`, which 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 string; otherwise, it returns NULL. We check for a valid pointer, compute the offset into the string, and use the offset to call the appropriate function. Thus, in only 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 run-time code is difficult to beat.

Timed task list. A third area in which 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. Rather, they only require that a number of tasks run at predetermined intervals. We can very simply handle this requirement, as shown in Listing 3.

In this listing, we define our own data type (`TIMED_TASK`), which 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 startup code, which must enable a periodic timer interrupt that increments the volatile variable `tick` at a fixed interval. We then enter the main loop.

The main loop checks for a non-zero `tick`, 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 the task to be executed, and if so, calls it via the function pointer.

If your application consists only of two or three tasks, then this approach is probably a bit of overkill. However, if your project has a large number of timed tasks, or you will likely have to add tasks in the future, you’ll find it

LISTING 3 Handling a timed task list

```
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))
                    (ptr->proc)(); /* Time to call the function */
            }
        }
    }
}
```

If you really feel the need to modify the array of pointers at run time, then go ahead and remove the `const` qualifier. But please don't try to sell me any products with your code in them!

rather palatable. Note that adding tasks and/or changing intervals simply requires editing the `timed_task[]` array. No code, per se, has to be changed.

Interrupt vectors. 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 on 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!

Listing 4 shows the vector table from the industrial power supply project I mentioned. This project was implemented using a Whitesmiths' compiler and a 68HC11.

A couple of points are worth making about Listing 4. First, the code is insufficient to locate the table correctly in memory. This would have to be done via linker directives.

Second, note that unused interrupts still have an entry in the table. Doing so ensures that the table is correctly aligned, and that traps can be placed on unexpected interrupts.

If any of these examples has whet your appetite for using arrays of function pointers, but you're still uncom-

fortable with the declaration complexity, fear not. Visit the *Embedded Systems Programming* Web site at www.embedded.com/code.htm, where you'll find a variety of declarations, ranging from the straightforward to the downright appalling. The examples are all reasonably practical in the sense that the desired functionality isn't 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 on the Web site adhere to conventions I've found to be useful over the years.

Function pointer arrays should always be declared static, and the scope of a function table should be highly localized. I've never come across a situation where it made sense to have a function table that wasn't declared this way.

Function pointer arrays should be declared `const`, which implies that the array cannot be modified after its initialization. However, if you really feel the need to modify the array of pointers at run time, then go ahead and remove the `const` qualifier. But please don't try to sell me any products with your code in them!

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

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

Then it may be invoked using either of these methods:

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

The advantage of the first method is an uncluttered syntax. However, it makes it look as if `fp` is a function, as opposed to being a function pointer.

LISTING 4 Vector table from the industrial power supply project

```
IMPORT VOID _stext();          /* startup routine 68HC11 specific*/

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          */
    I1OpC_Interrupt,        /* illegal      */
    _stext,                 /* cop fail     */
    _stext,                 /* cop clock fail */
    _stext,                 /* RESET       */
};
```

Typedefs work well when you regularly use a certain function pointer type because they save you from having to remember and type in the declaration.

Someone maintaining the code may end up searching in vain for the function `fp()`. With Method 2, we are clearly dereferencing a pointer. But when the declarations get complex, the added “*” can be a significant burden. Throughout the examples, both syntax varieties are shown. In practice, the latter syntax seems to be more popular, although I often use the former syntax.

One good way to reduce declaration complexity is to use typedefs. It is quite permissible to use a typedef to define a complex declaration, and to then use the new type like a simple type. Sticking with the example above, an alternative declaration would be:

```
typedef void (*PFV_I )(int);
PFV_I fp = fna; /* Declare a PFV_I
typed variable and init it */
fp(3);          /* Call
fna with 3 using method 1 */
(*fp)(3);      /* Call
fna with 3 using method 2 */
```

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 `fp` to be a variable of this type, and use it. Typedefs work well when you regularly use a certain function pointer type because they save you from having to remember and type in the declaration. The downside of using a typedef,

though, is the fact that it isn't 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.

If you prefer to use typedefs, I recommend that you use a consistent naming convention. My preference, which is adopted for the examples on the Web site, is as follows: 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`, and `D` are used. (Note the use of `S` (single) for float, to avoid confusion with `F` (function)). 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're on your own.

Note this 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.

Got that? Now make the most of those function pointers. **esp**

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

Nigel Jones is a consultant on all aspects of embedded development. He particularly enjoys working on small, distributed systems. He last wrote for ESP in November 1998, and enjoys hearing from readers at NAJones@compuserve.com.