

## ■ A 'C' Test: The 0x10 Best Questions for Would-be Embedded Programmers

Nigel Jones ■

An obligatory and significant part of the recruitment process for embedded systems programmers seems to be the 'C Test'. Over the years, I have had to both take and prepare such tests and in doing so have realized that these tests can be very informative for both the interviewer and interviewee. Furthermore, when given outside the pressure of an interview situation, they can also be quite entertaining (hence this article).

From the interviewee's perspective, you can learn a lot about the person that has written or administered the test. Is the test designed to show off the writer's knowledge of the minutiae of the ANSI standard rather than to test practical know-how? Does it test ludicrous knowledge, such as the ASCII values of certain characters? Are the questions heavily slanted towards your knowledge of system calls and memory allocation strategies, indicating that the writer may spend his time programming computers instead of embedded systems? If any of these are true, then I know I would seriously doubt whether I want the job in question.

From the interviewer's perspective, a test can reveal several things about the candidate. Primarily, one can determine the level of the candidate's knowledge of C. However, it is also very interesting to see how the person responds to questions to which they do not know the answers. Do they make intelligent choices, backed up with some good intuition, or do they just guess? Are they defensive when they are stumped, or do they exhibit a real curiosity about the problem, and see it as an opportunity to learn something? I find this information as useful as their raw performance on the test.

With these ideas in mind, I have attempted to construct a test that is heavily slanted towards the requirements of embedded systems. This is a lousy test to give to someone seeking a job writing compilers! The questions are almost all drawn from situations I have encountered over the years. Some of them are very tough; however, they should all be informative.

This test may be given to a wide range of candidates. Most entry-level applicants will do poorly on this test, while seasoned veterans should do very well. Points are not assigned to each question, as this tends to arbitrarily weight certain questions. However, if you choose to adapt this test for your own uses, feel free to assign scores.

### Preprocessor

1. *Using the #define statement, how would you declare a manifest constant that returns the number of seconds in a year? Disregard leap years in your answer.*

```
#define SECONDS_PER_YEAR (60UL * 60UL * 24UL * 365UL)
```

I'm looking for several things here:

- (a) Basic knowledge of the #define syntax (i.e. no semi-colon at the end, the need to parenthesize etc.).
- (b) A good choice of name, with capitalization and underscores.
- (c) An understanding that the pre-processor will evaluate constant expressions for you. Thus, it is clearer, and penalty free to spell out how you are calculating the number of seconds in a year, rather than actually doing the calculation yourself.

- (d) A realization that the expression will overflow an integer argument on a 16 bit machine—hence the need for the L, telling the compiler to treat the expression as a Long.
  - (e) As a bonus, if you modified the expression with a UL (indicating unsigned long), then you are off to a great start because you are showing that you are mindful of the perils of signed and unsigned types—and remember, first impressions count!
2. *Write the ‘standard’ MIN macro. That is, a macro that takes two arguments and returns the smaller of the two arguments.*

```
#define MIN(A,B) ((A) <= (B) ? (A) : (B))
```

The purpose of this question is to test the following:

- (a) Basic knowledge of the #define directive as used in macros. This is important, because until the inline operator becomes part of standard C, macros are the only portable way of generating inline code. Inline code is often necessary in embedded systems in order to achieve the required performance level.
- (b) Knowledge of the ternary conditional operator. This exists in C because it allows the compiler to potentially produce more optimal code than an if-then-else sequence. Given that performance is normally an issue in embedded systems, knowledge and use of this construct is important.
- (c) Understanding of the need to very carefully parenthesize arguments to macros.
- (d) I also use this question to start a discussion on the side effects of macros, e.g. what happens when you write code such as:

```
least = MIN(*p++, b);
```

3. *What is the purpose of the preprocessor directive #error?*

Either you know the answer to this, or you don't. If you don't, then see reference 1. This question is very useful for differentiating between normal folks and the nerds. It's only the nerds that actually read the appendices of C textbooks that find out about such things. Of course, if you aren't looking for a nerd, the candidate better hope she doesn't know the answer.

## Infinite Loops

---

4. *Infinite loops often arise in embedded systems. How does one code an infinite loop in C?*

There are several solutions to this question. My preferred solution is:

```
while(1)
{
...
}
```

Another common construct is:

```
for(;;)
{
...
}
```

Personally, I dislike this construct because the syntax doesn't exactly spell out what is going on. Thus, if a candidate gives this as a solution, I'll use it as an opportunity to explore their rationale for doing so. If their answer is basically—'I was taught to do it this way and I have never thought about it since'—then it tells me something (bad) about them. Conversely, if they state that it's the K&R preferred method and the only way to get an infinite loop passed Lint, then they score bonus points.

A third solution is to use a goto:

```
Loop:
```

```
...
goto Loop;
```

Candidates that propose this are either assembly language programmers (which is probably good), or else they are closet BASIC/FORTRAN programmers looking to get into a new field.

## Data declarations

---

5. *Using the variable `a`, write down definitions for the following:*

- (a) *An integer*
- (b) *A pointer to an integer*
- (c) *A pointer to a pointer to an integer*
- (d) *An array of ten integers*
- (e) *An array of ten pointers to integers*
- (f) *A pointer to an array of ten integers*
- (g) *A pointer to a function that takes an integer as an argument and returns an integer*
- (h) *An array of ten pointers to functions that take an integer argument and return an integer.*

The answers are:

- (a) `int a; // An integer`
- (b) `int *a; // A pointer to an integer`
- (c) `int **a; // A pointer to a pointer to an integer`
- (d) `int a[10]; // An array of 10 integers`
- (e) `int *a[10]; // An array of 10 pointers to integers`
- (f) `int (*a)[10]; // A pointer to an array of 10 integers`
- (g) `int (*a)(int); // A pointer to a function a that takes an integer argument and returns an integer`
- (h) `int (*a[10])(int); // An array of 10 pointers to functions that take an integer argument and return an integer`

People often claim that a couple of these are the sorts of thing that one looks up in textbooks—and I agree. While writing this article, I consulted textbooks to ensure the syntax was correct. However, I expect to be asked this question (or something close to it) when in an interview situation. Consequently, I make sure I know the answers—at least for the few hours of the interview. Candidates that don't know the answers (or at least most of them) are simply unprepared for the interview. If they can't be prepared for the interview, what will they be prepared for?

## Static

---

6. *What are the uses of the keyword `static`?*

This simple question is rarely answered completely. Static has three distinct uses in C:

- (a) A variable declared static within the body of a function maintains its value between function invocations.
- (b) A variable declared static within a module,<sup>1</sup> (but outside the body of a function) is accessible by all functions within that module. It is not accessible by functions within any other module. That is, it is a localized global.
- (c) Functions declared static within a module may only be called by other functions within that module. That is, the scope of the function is localized to the module within which it is declared.

Most candidates get the first part correct. A reasonable number get the second part correct, while a pitiful number understand answer (c). This is a serious weakness in a candidate, since they obviously do not understand the importance and benefits of

<sup>1</sup> Translation unit for the pedagogues out there.

localizing the scope of both data and code.

## Const

---

### 7. *What does the keyword `const` mean?*

As soon as the interviewee says ‘`const` means constant’, I know I’m dealing with an amateur. Dan Saks has exhaustively covered `const` in the last year, such that every reader of ESP should be extremely familiar with what `const` can and cannot do for you. If you haven’t been reading that column, suffice it to say that `const` means “read-only”. Although this answer doesn’t really do the subject justice, I’d accept it as a correct answer. (If you want the detailed answer, then read Saks’ columns—carefully!).

If the candidate gets the answer correct, then I’ll ask him these supplemental questions:

What do the following incomplete<sup>2</sup> declarations mean?

```
const int a;
int const a;
const int *a;
int * const a;
int const * a const;
```

The first two mean the same thing, namely `a` is a `const` (read-only) integer. The third means `a` is a pointer to a `const` integer (i.e., the integer isn’t modifiable, but the pointer is). The fourth declares `a` to be a `const` pointer to an integer (i.e., the integer pointed to by `a` is modifiable, but the pointer is not). The final declaration declares `a` to be a `const` pointer to a `const` integer (i.e., neither the integer pointed to by `a`, nor the pointer itself may be modified).

If the candidate correctly answers these questions, I’ll be impressed.

Incidentally, one might wonder why I put so much emphasis on `const`, since it is very easy to write a correctly functioning program without ever using it. There are several reasons:

1. The use of `const` conveys some very useful information to someone reading your code. In effect, declaring a parameter `const` tells the user about its intended usage. If you spend a lot of time cleaning up the mess left by other people, then you’ll quickly learn to appreciate this extra piece of information. (Of course, programmers that use `const`, rarely leave a mess for others to clean up...)
2. `const` has the potential for generating tighter code by giving the optimizer some additional information.
3. Code that uses `const` liberally is inherently protected by the compiler against inadvertent coding constructs that result in parameters being changed that should not be. In short, they tend to have fewer bugs.

## Volatile

---

### 8. *What does the keyword `volatile` mean? Give three different examples of its use.*

A `volatile` variable is one that can change unexpectedly. Consequently, the compiler can make no assumptions about the value of the variable. In particular, the optimizer must be careful to reload the variable every time it is used instead of holding a copy in a register. Examples of `volatile` variables are:

- (a) Hardware registers in peripherals (e.g., status registers)
- (b) Non-stack variables referenced within an interrupt service routine.
- (c) Variables shared by multiple tasks in a multi-threaded application.

---

<sup>2</sup> I’ve had complaints that these code fragments are incorrect syntax. This is correct. However, writing down syntactically correct code pretty much gives the game away.

If a candidate does not know the answer to this question, they aren't hired. I consider this the most fundamental question that distinguishes between a 'C programmer' and an 'embedded systems programmer'. Embedded folks deal with hardware, interrupts, RTOSes, and the like. All of these require `volatile` variables. Failure to understand the concept of `volatile` will lead to disaster.

On the (dubious) assumption that the interviewee gets this question correct, I like to probe a little deeper, to see if they really understand the full significance of `volatile`. In particular, I'll ask them the following:

(a) *Can a parameter be both `const` and `volatile`? Explain your answer.*

(b) *Can a pointer be `volatile`? Explain your answer.*

(c) *What is wrong with the following function?*

```
int square(volatile int *ptr)
{
    return *ptr * *ptr;
}
```

The answers are as follows:

- (a) Yes. An example is a read only status register. It is `volatile` because it can change unexpectedly. It is `const` because the program should not attempt to modify it.
- (b) Yes. Although this is not very common. An example is when an interrupt service routine modifies a pointer to a buffer.
- (c) This one is wicked. The intent of the code is to return the square of the value pointed to by `*ptr`. However, since `*ptr` points to a `volatile` parameter, the compiler will generate code that looks something like this:

```
int square(volatile int *ptr)
{
    int a,b;
    a = *ptr;
    b = *ptr;
    return a * b;
}
```

Since it is possible for the value of `*ptr` to change unexpectedly, it is possible for `a` and `b` to be different. Consequently, this code could return a number that is not a square! The correct way to code this is:

```
long square(volatile int *ptr)
{
    int a;
    a = *ptr;
    return a * a;
}
```

## Bit Manipulation

9. *Embedded systems always require the user to manipulate bits in registers or variables. Given an integer variable `a`, write two code fragments. The first should set bit 3 of `a`. The second should clear bit 3 of `a`. In both cases, the remaining bits should be unmodified.*

These are the three basic responses to this question:

- (a) No idea. The interviewee cannot have done any embedded systems work.

- (b) Use bit fields. Bit fields are right up there with trigraphs as the most brain-dead portion of C. Bit fields are inherently non-portable across compilers, and as such guarantee that your code is not reusable. I recently had the misfortune to look at a driver written by Infineon for one of their more complex communications chip. It used bit fields, and was completely useless because my compiler implemented the bit fields the other way around. The moral—never let a non-embedded person anywhere near a real piece of hardware!<sup>3</sup>
- (c) Use #defines and bit masks. This is a highly portable method, and is the one that should be used. My optimal solution to this problem would be:

```
#define BIT3      (0x1 << 3)
static int a;
void set_bit3(void) {
    a |= BIT3;
}
void clear_bit3(void) {
    a &= ~BIT3;
}
```

Some people prefer to define a mask, together with manifest constants for the set & clear values. This is also acceptable. The important elements that I'm looking for are the use of manifest constants, together with the |= and &= ~ constructs.

## Accessing fixed memory locations

---

10. *Embedded systems are often characterized by requiring the programmer to access a specific memory location. On a certain project it is required to set an integer variable at the absolute address 0x67a9 to the value 0xaa55. The compiler is a pure ANSI compiler. Write code to accomplish this task.*

This problem tests whether you know that it is legal to typecast an integer to a pointer in order to access an absolute location. The exact syntax varies depending upon one's style. However, I would typically be looking for something like this:

```
int *ptr;
ptr = (int *)0x67a9;
*ptr = 0xaa55;
```

A more obfuscated approach is:

```
*(int * const) (0x67a9) = 0xaa55;
```

Even if your taste runs more to the second solution, I suggest the first solution when you are in an interview situation.

## Interrupts

---

11. *Interrupts are an important part of embedded systems. Consequently, many compiler vendors offer an extension to standard C to support interrupts. Typically, this new key word is \_\_interrupt. The following code uses \_\_interrupt to define an interrupt service routine. Comment on the code.*

```
__interrupt double compute_area(double radius) {
{
    double area = PI * radius * radius;
    printf("\nArea = %f", area);
```

---

<sup>3</sup> I've recently softened my stance on bit fields. At least one compiler vendor (IAR) now offers a compiler switch for specifying the bit field ordering. Furthermore the compiler generates optimal code with bit field defined registers – and as such I now do use bit fields in IAR applications.

```
return area;
}
```

This function has so much wrong with it, it's almost tough to know where to start.

- (a) Interrupt service routines cannot return a value. If you don't understand this, then you aren't hired.
- (b) ISR's cannot be passed parameters. See item (a) for your employment prospects if you missed this.
- (c) On many processors / compilers, floating point operations are not necessarily re-entrant. In some cases one needs to stack additional registers, in other cases, one simply cannot do floating point in an ISR. Furthermore, given that a general rule of thumb is that ISRs should be short and sweet, one wonders about the wisdom of doing floating point math here.
- (d) In a similar vein to point (c), printf() often has problems with reentrancy and performance. If you missed points (c) & (d) then I wouldn't be too hard on you. Needless to say, if you got these two points, then your employment prospects are looking better and better.

## Code Examples

---

### 12. *What does the following code output and why?*

```
void foo(void)
{
    unsigned int a = 6;
    int b = -20;
    (a+b > 6) ? puts("> 6") : puts("<= 6");
}
```

This question tests whether you understand the integer promotion rules in C – an area that I find is very poorly understood by many developers. Anyway, the answer is that this outputs “> 6”. The reason for this is that expressions involving signed and unsigned types have all operands promoted to unsigned types. Thus -20 becomes a very large positive integer and the expression evaluates to greater than 6. This is a very important point in embedded systems where unsigned data types should be used frequently (see reference 2). If you get this one wrong, then you are perilously close to not being hired.

### 13. *Comment on the following code fragment?*

```
unsigned int zero = 0;
unsigned int compzero = 0xFFFF; /*1's complement of zero */
```

On machines where an int is not 16 bits, this will be incorrect. It should be coded:

```
unsigned int compzero = ~0;
```

This question really gets to whether the candidate understands the importance of word length on a computer. In my experience, good embedded programmers are critically aware of the underlying hardware and its limitations, whereas computer programmers tend to dismiss the hardware as a necessary annoyance.

By this stage, candidates are either completely demoralized—or they are on a roll and having a good time. If it is obvious that the candidate isn't very good, then the test is terminated at this point. However, if the candidate is doing well, then I throw in these supplemental questions. These questions are hard, and I expect that only the very best candidates will do well on them. In posing these questions, I'm looking more at the way the candidate tackles the problems, rather than the answers. Anyway, have fun...

## Dynamic memory allocation.

---

### 14. *Although not as common as in non-embedded computers, embedded systems still do dynamically allocate*



*memory from the heap. What are the problems with dynamic memory allocation in embedded systems?*

Here, I expect the user to mention memory fragmentation, problems with garbage collection, variable execution time, etc. This topic has been covered extensively in ESP, mainly by Plauger. His explanations are far more insightful than anything I could offer here, so go and read those back issues! Having lulled the candidate into a sense of false security, I then offer up this tidbit:

*What does the following code fragment output and why?*

```
char *ptr;
if ((ptr = (char *)malloc(0)) == NULL) {
    puts("Got a null pointer");
}
else {
    puts("Got a valid pointer");
}
```

This is a fun question. I stumbled across this only recently, when a colleague of mine inadvertently passed a value of 0 to malloc, and got back a valid pointer! After doing some digging, I discovered that the result of malloc(0) is implementation defined, so that the correct answer is ‘it depends.’ I use this to start a discussion on what the interviewee thinks is the correct thing for malloc to do. Getting the right answer here is nowhere near as important as the way you approach the problem and the rationale for your decision.

## Typedef

---

15. *Typedef is frequently used in C to declare synonyms for pre-existing data types. It is also possible to use the preprocessor to do something similar. For instance, consider the following code fragment:*

```
#define dPS struct s *
typedef struct s * tPS;
```

*The intent in both cases is to define dPS and tPS to be pointers to structure s. Which method (if any) is preferred and why?*

This is a very subtle question, and anyone that gets it right (for the right reason) is to be congratulated or condemned (“get a life” springs to mind). The answer is the typedef is preferred. Consider the declarations:

```
dPS p1,p2;
tPS p3,p4;
```

The first expands to

```
struct s * p1, p2;
```

which defines p1 to be a pointer to the structure and p2 to be an actual structure, which is probably not what you wanted. The second example correctly defines p3 & p4 to be pointers.

## Obfuscated syntax

---

16. *C allows some appalling constructs. Is this construct legal, and if so what does this code do?*

```
int a = 5, b = 7, c;
c = a+++b;
```

This question is intended to be a lighthearted end to the quiz, as, believe it or not, this is perfectly legal syntax. The question is how does the compiler treat it? Those poor compiler writers actually debated this issue, and came up with the “maximum munch” rule, which stipulates that the compiler should bite off as big a (legal) chunk as it can. Hence, this code is treated as:

```
c = a++ + b;
```

Thus, after this code is executed, a = 6, b = 7 & c = 12;

If you knew the answer, or guessed correctly—then well done. If you didn’t know the answer then I would not consider this to be a problem. I find the biggest benefit of this question is that it is very good for stimulating questions on coding styles, the



value of code reviews and the benefits of using lint.

Well folks, there you have it. That was my version of the C test. I hope you had as much fun doing it as I had writing it. If you think the test is a good test, then by all means use it in your recruitment. Who knows, I may get lucky in a year or two and end up being on the receiving end of my own work.

References:

1. In Praise of the `#error` directive. ESP September 1999.
2. Efficient C Code for Eight-Bit MCUs. ESP November 1988.

## ■ Minimize Interrupt Service Routine Overhead

Nigel Jones ■

With all the automation available today, it's easy for programmers to overlook costly overhead introduced into machine code by the compiler. Interrupt handlers are one key area worthy of a closer inspection.

In the early days of embedded C compilers, interrupt service routines (ISRs) had to be written in assembly language. Today, most compilers let the developer identify a function as an ISR, with the compiler taking care of all of the hassles associated with the ISR. This could include placing the correct entry into the interrupt vector table, stacking and unstacking registers, and terminating the function with a return from interrupt instruction. There are even compilers available that know when an ISR needs to clear a particular flag before the ISR terminates and will insert the proper code.

Although these advances are most welcome, they have come at a price—namely that it's so easy to write an interrupt handler in a high-level language that one can easily lose track of the overhead that the compiler is introducing. Doing so can unwittingly result in a high price for the convenience of using a high-level language.

ISR overhead can be split into two parts, fixed and variable. Fixed overhead is associated with the CPU detecting that an interrupt has occurred, vectoring to the ISR, clearing any required flags to prevent the ISR from being re-entered, and finally executing the return from interrupt instruction. Thus, an ISR that does nothing, such as the one shown in Listing 1, still incurs some CPU overhead every time it occurs. Naturally, there isn't much a compiler can do about the fixed overhead.

```
__interrupt void timer_isr(void)
{
    /* Do nothing */
}
```

Listing 1. Do nothing ISR

ISR variable overhead is the time spent stacking and unstacking the registers used in the ISR. This is a key topic, as it's not uncommon for the time spent stacking and unstacking registers to dwarf the time spent doing useful work. One of the ironies of using modern, register-rich CPUs is that the presence of all those registers encourages the compiler writers to use them! Normally, this leads to very fast, tight code. However in an ISR, all those registers can work to our detriment.

Consider an ISR that uses 12 registers (and thus incurs 12 push and 12 pop operations). Even though the ISR's body may be faster than code that uses just six registers, the overall execution time of the ISR may increase simply because it must stack and unstack so many registers. For infrequently occurring interrupts, this is irrelevant. However for interrupts that occur frequently, this overhead can rapidly consume a very large portion of the CPU's bandwidth.

For example, consider a full-duplex serial link running at 38,400 baud on a CPU with a 1-MHz instruction cycle. If the CPU is interrupted upon receipt and transmission of every character, then assuming 10 bits per byte, it'll be interrupted every  $106 / (38,400 / 10) / 2 = 130$   $\mu$ s. If each interrupt requires 12 registers to be pushed and popped, then the processor spends  $24/130 = 18\%$  of its time doing nothing more than stack operations. A coding change that requires only six registers to be stacked would free up 9% of the CPU time and save stack space.

To determine how much overhead you are incurring in an ISR, have the compiler generate an assembly language listing of your C code, preferably with the C instructions interleaved. An example is shown in Listing 2.

```

160 static __interrupt void timer0_CompareMatchAIsr(void)
\   timer0_CompareMatchAIsr:
161 {
\   00000000  938A          ST      -Y, R24
\   00000002  93FA          ST      -Y, R31
\   00000004  93EA          ST      -Y, R30
\   00000006  923A          ST      -Y, R3
\   00000008  922A          ST      -Y, R2
\   0000000A  921A          ST      -Y, R1
\   0000000C  920A          ST      -Y, R0
\   0000000E  937A          ST      -Y, R23
\   00000010  936A          ST      -Y, R22
\   00000012  935A          ST      -Y, R21
\   00000014  934A          ST      -Y, R20
\   00000016  933A          ST      -Y, R19
\   00000018  932A          ST      -Y, R18
\   0000001A  931A          ST      -Y, R17
\   0000001C  930A          ST      -Y, R16
\   0000001E  B78F          IN       R24, 0x3F
162 TCCR0B = 0; /* Stop the timer */
\   00000020  E000          LDI      R16, 0
\   00000022  BF03          OUT      0x33, R16
163 fifo_AddEvent(Event); /* Post the event */
\   00000024  9100....      LDS      R16, Event
\   00000028  ....          RCALL    fifo_AddEvent
164 }
\   0000002A  BF8F          OUT      0x3F, R24
\   0000002C  9109          LD       R16, Y+
\   0000002E  9119          LD       R17, Y+
\   00000030  9129          LD       R18, Y+
\   00000032  9139          LD       R19, Y+
\   00000034  9149          LD       R20, Y+
\   00000036  9159          LD       R21, Y+
\   00000038  9169          LD       R22, Y+
\   0000003A  9179          LD       R23, Y+
\   0000003C  9009          LD       R0, Y+
\   0000003E  9019          LD       R1, Y+
\   00000040  9029          LD       R2, Y+
\   00000042  9039          LD       R3, Y+
\   00000044  91E9          LD       R30, Y+
\   00000046  91F9          LD       R31, Y+
\   00000048  9189          LD       R24, Y+
\   0000004A  9518          RETI

```

Listing 2. Mixed C and generated assembly language listing of two-line ISR

Even if you aren't familiar with your CPU's instruction set, the push and pop operations are usually easy to spot. Note that this two-line ISR requires 15 registers to be stacked and unstacked.

I'll present various techniques for minimizing ISR overhead. The order they're presented in follows a project's progression. The first suggestions are only applicable at the start of a project, whereas the final suggestions apply to the poor soul at the end of a project who needs to improve performance without changing anything.

## CPU selection

If you're a regular reader of *Embedded Systems Design*, you know that the latest CPUs offer amazing capabilities; yet survey after survey shows that huge numbers of us continue to work with ancient architectures such as the 8051. I previously wrote this off as a simple case of inertia until I went through an interesting exercise about a year ago, when I had to select a small CPU core to be embedded in an ASIC. The ASIC was to be used in a portable product where ultra low-power consumption was critical. As is the case for most low-power products, most of the code was to be executed under interrupt, and hence ISR overhead was a major concern.

To determine the best CPU for the application, example pieces of code were compiled using the best compilers I could find for each CPU candidate and the energy consumed by each CPU calculated. The results were quite clear: the older CPUs such as the 8051 and the HC08 from Freescale had a distinct advantage over the newer, register-rich architectures. This advantage was due in part to the low ISR overhead of these devices. In the case of the 8051, this advantage also had a lot to do with the fact that the 8051 allows one to allocate a register bank to a particular block of code—such as an ISR—and thus drastically reduce the number of registers that need to be stacked.

The bottom line is if you know your code will be dominated by interrupt handlers, don't be in too much of a hurry to embrace the latest and greatest register-rich CPU.

## Compiler selection

As will be shown shortly, a major factor on ISR performance is your compiler's register allocation strategy. If you aren't familiar with this term, it refers to the algorithm the compiler uses in allocating registers across function calls. In broad terms, a compiler has three options:

- Assume that a called function will trash every register
- Require that the called function preserve every register that it uses
- Use a hybrid approach, whereby certain registers are deemed scratch registers—and hence may be trashed by a called function, and other registers are preserved registers—and thus must be preserved if used by a called function

Now, if your compiler has a sophisticated global register-coloring algorithm, whereby it can track register allocation across inter-module function calls—and so eliminate all unnecessary register preservation instructions, then these three algorithms effectively all collapse down to the same thing, and thus the strategy is of little concern. However, if your compiler doesn't do this, then you need to be aware of its impact on making function calls from within interrupt handlers.

## Firmware design

Once you've chosen your CPU and compiler, you'll probably turn your attention to the detailed firmware architecture. A key decision that you'll have to make is what interrupts are needed and what gets done within them. In making these decisions, be aware of the consequences of making function calls from within an ISR.

Consider the example shown in Listing 3, which is the source code for the assembly language listing shown in Listing 2. On the face of it, it meets the criteria of keeping interrupt handlers short and simple. After all it's just two lines of code.

```
void fifo_AddEvent(uint8_t event);

__interrupt void timer_isr(void)
{
    TCCROB = 0;                /* Stop the timer */
    fifo_AddEvent(Event);      /* Post the event */
}
```

Listing 3. A “simple” ISR

However, as Listing 2 shows, a huge number of registers have been stacked. The reason is quite simple: the compiler that generated this code uses a hybrid register allocation strategy. Thus, without any other information, the compiler has to assume that the called function will use all the scratch registers it's allowed to use—and so stacks all of them. Now, if the called function is a complex function, then stacking all the registers is unfortunate but necessary. However, if the called function is simple, then most of the register stacking is probably unnecessary. In order to avoid this problem, don't make function calls from within an ISR.

This advice falls into the same category as soldiers being advised to avoid dangerous situations for health reasons. It's true, but useless most of the time. Presumably, if you're making a function call, you're doing it for a good reason. In which case, what can you do about it? In a nutshell, you must do something that lets the compiler know what registers the called function is using. There are three ways you can do this, but they all amount to the same thing.

If you are writing in C++, or C99, then make the function that's called from the ISR an inline function. This will result in copies of the function being used throughout the project, but should result in only the required registers being stacked. It's your call as to whether the tradeoff is acceptable.

If you are using C89, you can do something similar by using macros and global variables—with all the problems that this entails.

The third option is to ensure that the ISR and the called function reside in the same module. In this case, a good compiler can see exactly what you're calling and thus stack only the required registers. Furthermore, if it's possible to declare the called function as static, there's a good chance that the compiler will inline the function and also eliminate the overhead of the actual function call.

## Occasional function calls

---

Although the overhead of a function call can be annoying, it can be frustrating when you have code that looks like Listing 4. In this case, most of the time, the ISR simply decrements a counter. However, when the counter reaches zero, a function call is made. Unfortunately, many compilers will look at this ISR, see that a function call is made, and stack all the registers for the function call every time the ISR is invoked. This wastes CPU cycles. The long-term solution to this problem is to pressure the compiler vendors to improve their product such that the registers only get stacked when the function call is made. In the short term, use a software interrupt. The code for such an approach is shown in Listing 5.

```
__interrupt void timer_isr(void)
{
    if (--Cntr == 0)
    {
        post_event(TIMER_TICK);
    }
}
```

Listing 4. An occasional function call

```
__interrupt void timer_isr(void)
{
    if (--Cntr == 0)
    {
        __software_interrupt;
    }
}

__interrupt void software_isr(void)
{
    post_event(TIMER_TICK);
}
```

Listing 5. Use of software interrupt

In place of the function call, we generate a software interrupt. This results in another interrupt being generated. The requisite function call is made from within this interrupt. The advantage of this approach is that the timer ISR no longer needs to stack the registers associated with the function call. Of course the software interrupt handler does need to stack the registers, but now

they're only stacked when they're really needed. The main cost of this approach is slightly increased latency from the time the interrupt occurs to the time the function call is made.

Depending on your perspective, this technique is either very elegant or a kludge. However, with adequate documentation, it should be possible to explain clearly what you're doing and why. For me, the choice between doing this and resorting to assembly language is clear cut, but the choice is yours.

Incidentally, you may be unaware of the concept of a software interrupt. Some CPUs include in their instruction set a software interrupt instruction (SWI). Execution of the instruction causes an interrupt flag to be set (just as if a timer had rolled over, or a character had been received in a UART). If all the usual criteria are met, at some point the CPU will vector to the software interrupt handler and execute it just like any normal hardware based interrupt.

Don't be despondent if your CPU doesn't include a SWI because it's possible to fashion a software interrupt in several ways using unused hardware resources. The easiest way is typically to configure an unused port pin as an output, while also configuring it to interrupt on change. Thus, simply toggling the port pin will generate an interrupt.

A second technique is to employ a spare counter. In this case, one can simply load the counter with its maximum value, configure it to interrupt on roll over, then enable the counter to count up. With a bit of ingenuity, you'll find that most of the hardware resources in a typical microcontroller can be configured to generate an interrupt upon software command.

A final warning for those of you whose CPU supports an SWI instruction, is that debuggers often make use of the SWI instruction to set breakpoints. So before using this technique, make sure you aren't going to break your debugger.

## **Coding constructs**

---

What if your CPU has been selected and your software architecture designed, such that the functionality of your interrupt handlers is set? In this case, it's time to look at some of your coding constructs. Small changes in this area can make surprisingly large changes to the number of registers that need to be stacked. Most of the normal suggestions for making code tighter apply, of course, and won't be reiterated here. However, there are a few areas to watch out for.

Switch statements are a bad idea for two reasons. First, the overhead associated with a switch statement is typically quite large, resulting in a lot of registers being stacked. Second, most compilers will implement switch statements in one of several ways depending on the values that are switched. Thus, a small change to a switch statement can result in a completely different set of instructions being generated (and hence registers stacked). The result is that an innocuous change to the code can make a dramatic difference to the CPU bandwidth consumed. This is not something that makes for a maintainable system.

Watch out for unexpected type promotions. These can have a devastating impact on an ISR in two ways. First, a type promotion can result in more registers being used to hold a variable than is necessary (which of course means that more registers need to be stacked). Second, a type promotion may result in the compiler making a function call to a library function to perform an operation rather than doing it in line. If you're lucky, your compiler knows what registers are used by library functions. If you're unlucky, you'll be hit with the full overhead of a function call.

Interrupts often test and/or set flags. Many processors contain special flag locations that can be operated on without going through a CPU register. These special flag locations can thus have a double benefit within interrupts as one avoids the overhead of stacking the intermediate registers.

If you have done all that you can to minimize overhead, and you are still facing an unacceptable burden, before reaching for the assembler, look at your compiler optimization settings, as the usual rules don't always apply to interrupt handlers.

Most compilers will let you optimize for size or speed. Furthermore, most compiler manuals carry a notice that sometimes optimizing for speed will actually result in a smaller footprint than optimizing for size. This can occur because the act of inlining code, unraveling loops, and so forth, results in the optimizer seeing a different view of the code and making optimizations that weren't previously possible. Interestingly, I've never seen a compiler manual admonish the user that

optimizing for size may result in faster executing code. However, this is exactly what can happen in the case of an ISR.

I'm not privy to the way compiler writers optimize interrupt handlers. However, what seems to happen is that the optimizer concentrates on optimizing the C language constructs and then simply adjusts the registers that need to be stacked based on the resulting code, rather than looking at the entire routine. By asking the compiler to optimize for size, you create the possibility of the compiler generating more compact code that uses less registers. Because it uses less registers, the stacking overhead is reduced and the overall execution time of the ISR is potentially reduced. Thus the bottom line is, when asking a compiler to optimize an ISR, try both optimizing for speed and size, as you may find that size-optimized code is actually faster.

---

This article was published in the January 2007 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. "Minimize your ISR overhead" Embedded Systems Programming, January 2007.



## ■ Use Strings to Internationalize C Programs

Nigel Jones ■

Products destined for use in multiple countries often require user interfaces that support several human languages. Sloppy string management in your programs could result in unintelligible babble.

A decade or two ago, most embedded systems had an extremely limited user interface. In most cases, the interface was either non-existent, or it consisted of a few LEDs and the odd jumper or push button. As the cost of display technology has plummeted, alphanumeric user interfaces have become increasingly common. Simultaneously, a variety of technological, economic, and political pressures have brought about the need for products to be sold in many countries. As a result, the need for an embedded system to support multiple languages has become apparent.

This problem is even more acute for the non-embedded computer world. Part of the solution for that marketplace was the introduction of Unicode, wide character types, and so on. Unfortunately, these techniques require storage capabilities and display resolutions rarely found in embedded systems. Instead, most embedded systems with displays typically use a low-resolution LCD or vacuum fluorescent display (VFD) with a built-in character generator. It's this type of display that I'll be concentrating on.

### **Lessons learned**

A few years ago, I worked on an industrial measurement system. The product was to be sold in both North America and Europe. Consequently, one of the essential design requirements was to support multiple European languages. The product in question has a 240 x 128-pixel LCD panel with a built-in character generator. The character generator contains a subset of the ASCII 256-character set, including "specialized" characters such as c, u, and e.

Anyway, as I pondered possible approaches to the design, I looked back at my previous attempts to solve this problem. They weren't pretty! At the risk of ruining what little reputation I may have gained, I think it's instructive to look at these previous attempts.

#### *Lesson 1*

The first product that I designed that had an alphanumeric display was implemented with the typical arrogance of a native English speaker. Namely, it never occurred to me to even consider the rest of the world. As a result, my code was littered with text strings. That is, you'd see the assembly language (yes, it was that long ago) equivalent of:

```
WtStr("Jet Black");
```

To make matters worse, this construct would be found in many functions, split between several files.

The foolishness of this approach struck home when I was asked to produce a version of the product for the German market. I quickly realized that I had to edit source code files to implement the translation. This had several ramifications:

- A separate make environment was needed for each language.
- Every time a change had to be made to the code, the same modification had to be made to each language's version of the file. In short, it was a maintenance nightmare.
- The source code had to be given to the translator. I think you can imagine the problems this caused.

## Lesson 2

The next product was a big improvement, because I did what should have been done in the first place, which was to place all the text strings into one file. That is, there were alternative string files called `english.c`, `german.c`, and so on for additional languages, each containing all of the strings for a particular language. Each string file contained a single array definition that looked something like this:

```
char const * const strings[] =
{
    "Jet Black",
    "Red",
    .....
};
```

Thus, to display a particular string, my code looked like this:

```
WrStr(strings[0]);
```

Now all I had to do to enable support for a new language was to hand a copy of `english.c` to the translator, and have him produce the equivalent strings for the new language. Unfortunately, it didn't work out that way. It turns out that the English language is extraordinarily terse when compared with certain other languages. For example, the German equivalent of Jet Black is Rabenschwarz.

Working from `english.c`, the translator assumed there were just nine characters into which to fit the translation. Thus, the translator was forced into abbreviating the German. However, in many cases, there was actually more space available on the display such that the abbreviation looked awkward in the product. The only way to find out was to execute the code and look at the results. This is a lot harder than it sounds, because many strings are only displayed in exceptional conditions. Thus one has to generate all the external stimuli such that those exceptions occur. In short, the translation effort remained a Herculean task.

Once it was complete, I still wasn't out of the woods, because the different length strings caused the memory allocation to change significantly. Although it did not happen, theoretically I could have run out of memory.

## Lesson 3

By the time I was working on my third product requiring multiple language support, I was a lot wiser and memory capacities had increased dramatically. As a result, I now ensured that every string in my string file was padded with spaces out to the maximum allowed field width. Furthermore, I had also learned the intricacies of conditional compilation and passing command line arguments to make, such that I included every language into the same text file. Thus, `strings.c` looked something like Listing 1.

```
#if defined(ENGLISH)
char const * const strings[] =
{
    "Jet Black ",
    ...
    "Good Bye ",
    ...
    "Evening"
};
#elif defined(GERMAN)
char const * const strings[] =
{
    "Rabenschwarz",
    ...
```

```

    "Auf Wiedersehen ",
    ...
    "Abend "
};
#endif

```

#### Listing 1. Multiple languages in a single C module

This third solution worked well, except that at the same time, the size of the alphanumeric displays and the complexity of the user interface had increased. While my first product had just 30 or 40 strings, this latest product had around 500. Thus, the bulk of the user interface code ended up looking like this:

```

WrStr(strings[27]);
WrStr(strings[47]);
WrStr(strings[108]);

```

This code doesn't make clear what string I was actually displaying. So I was beginning to long for the original:

```

WrStr("Jet Black");

```

I ran into another major problem at this time. As the product evolved, so did the strings. I found myself wanting to delete certain strings that were no longer needed. But I couldn't do that without destroying my indexing scheme. Thus, I was forced into changing unwanted text into null strings, such that strings.c now contained sequences like this:

```

char const * const strings[] = {
    "Jet Black ",
    ...
    "", /* deleted */
    ...
    "Evening"
};

```

Although this saved the space consumed by the string, I was still wasting ROM on the pointer. In addition, it looked ugly and had "kludge" written all over it. I also ran into a more serious problem. From a maintenance perspective, it would be very useful if related strings were in contiguous locations. Thus if a particular field could contain "Jet Black," "Red," or "Pale Grey," I would place these together in the string file. However, two years later, when marketing asked for "Yellow" to be added to the list of selections, I was forced to place "Yellow" at the end of the string list, well away from its partners. This pained me greatly.

There was one final problem with this implementation (and all the others) and that was the fact that the strings array was a global. I've become quite paranoid about globals in the last few years, such that when I look back at the code now, I have to confess that I cringe.

I also discovered a neat feature that was missing from all of the previous disasters. A few years ago, I saw a product demonstration in which the language was changeable on the fly. That is, without changing the operating mode or cycling power, the entire user interface could be changed to another language. The demonstration was incredibly slick. (Consider the following scenario. Your product is being introduced at a trade show. Some native French speakers come to the booth to look at the product. With the push of a button, you switch the user interface to French. You're already halfway to a sale.)

In addition to its value as a sales tool, the ability to change language on the fly is also a valuable tool for validating a new translation. It's particularly useful when the correct translation of a word depends heavily upon its context. When working through the string file, the translator can't see the context, so having the ability to operate the product and switch back and forth between languages is invaluable.

## An international approach

Being quite a few years wiser than when Lesson #3 was learned, I was determined to come up with a scheme that would address as many of the aforementioned problems as possible and add the ability to switch languages on the fly. What follows is my solution. It's not perfect-but it is a lot better than any of the previous attempts.

The first decision I made was to separate the string retrieval mechanism and the string storage technique. There would be no more global strings array. Instead, strings would be accessed through a function call. This access function would take a string number as an argument and return a pointer to the desired string. Its provisional prototype is:

```
char const * strGet(int string_no);
```

This abstraction gave me freedom to implement the data storage part of the strings in whatever manner I saw fit. In particular, I realized that implementing the storage as an array of data structures would have considerable benefit. My data structure looked like this:

```
#define N_LANG4
typedef struct
{
    /*
     * Maximum length
     */
    int const len;
    /*
     * Array of pointers to language-specific string
     */
    char const * const text[N_LANG];
} STRING;
```

This arrangement offered some serious benefits. First, the maximum allowed string length for the field is stored with the text. Second, the original string and all of the various translations of it are together in one place. This makes life a lot easier for the translator. The downside is, of course, that this uses a lot more ROM than a preprocessor-based selection. In my case, I had the ROM to spare. With this data structure in hand, the strings array now looked like Listing 2.

```
static const STRING strings[] =
{
    {
        15,
        {
            "Jet Black ",          /* English */
            "Rabenschwarz ",      /* German */
            ...
        }
    },
    {
        15,
        {
            "Red ",                /* English */
            "Rot ",                /* German */
            ...
        }
    },
    ...
};
```

Listing 2. A better string storage structure

The access function also had another valuable property. When the application requested a string, the access function could interrogate the relevant database to find out the current language, and return the requisite pointer. Voila! Language-independent code. The access function looks something like this:

```
char const * strGet(int str_no)
{
    return strings[str_no].text[GetLanguage()];
}
```

## Further improvements

---

The previous approach certainly solved a few of my problems. However, it did nothing for the problems of indecipherable string numbers or adding and deleting strings in the middle of the array. I needed a means of referring to the strings in a meaningful manner, together with a way of automatically re-indexing the string table. My solution was to use an enumerated type. The members of the enumeration are given the same name as the strings to which they refer. An example should help clarify this.

Assume the first four strings that appear in the strings array are “Jet Black,” “Red,” “Pale Grey,” and “Yellow.” To display “Red,” I would have to call:

```
WrStr(strGet(1));
```

Instead, I define an enumerated list as follows:

```
typedef enum { JET_BLACK, RED, PALE_GREY, YELLOW, ... } STR;
```

I now change the prototype of `strGet()` to (with the changes in red):

```
char const * strGet(STR str_no);
```

Thus, to display the string “Red,” the code becomes:

```
WrStr(strGet(RED));
```

Furthermore, by defining a macro, `Wrstr(X)` as follows,

```
#define Wrstr(X) WrStr(strGet((X)))
```

we can write:

```
Wrstr(RED);
```

This is just as readable as the original `WrStr("Red")`, but without any of the aforementioned problems. Furthermore, this technique allows one to insert or delete strings at will. For instance, I could insert “Pink” before “Red” in the strings array, do the same in the enumeration list, and recompile-and none of the code should be broken.

This was a major step forward, because I now had a system that met most of my goals

- No global data
- Easy to add additional languages
- Language selection on the fly
- Code is meaningful
- Allows strings to be inserted and deleted at will

## Gotchas of enumerated types

---

However, a couple of “gotchas” arise when using enumerated types in this way. The first, and most important, is portability. ANSI C requires only that the first 31 identifiers in a name be significant. If you can guarantee that all of your strings are shorter than this, there is no problem. If you cannot make that guarantee, these are some of your options:

- Switch to a compiler that allows unlimited identifier lengths. Many compilers do have this feature.
- Ensure that all strings are unique within the first 31 characters. Note that if they aren’t, the compiler should issue a re-declaration warning.

The next issue to watch out for occurs when you have a large number of strings. ANSI allows the compiler writer to implement enumerated types using an implementation-defined integer type. Thus, technically speaking, a compiler could limit the number of items in an enumeration to 127 (the largest positive number that can fit into an 8-bit integer). Thus, if you have rigid portability constraints, this technique may be problematic. However, practically speaking, most compilers appear to implement enumerations either as an int, or as the smallest integral type that can accommodate the enumerated list.

The third problem I ran into concerns the limited number of legal characters that can make up an identifier (that is, a-z, A-Z, 0-9, and \_). For instance, it is impossible to exactly reproduce the string “3 Score & 10!”. In situations like this, I used \_ wherever I couldn’t make the exact substitution. Thus, the enumerated value for “3 Score & 10!” would be \_3\_SCORE\_\_10\_, or possibly \_3\_SCORE\_AND\_10\_. This isn’t perfect, but it’s still better than a meaningless identifier such as STRING\_49.

The final issue was the absolute necessity of keeping the string file and the enumerated list synchronized. This proved to be quite difficult. To aid the process, I modified the string table slightly to include the enumerated type name in the comment field. Next, I ensured that the last entry in the enumerated list was always LAST\_STR. This allowed the string array to be changed from being an incomplete declaration to a complete declaration. This means that the compiler will complain if the number of elements in the enumerated list does not exactly match the number of elements in the string array. This proved to be valuable in keeping the two files synchronized.

### *The winning design*

The final enumerated list and string table declarations are as shown in Listing 3.

```
typedef enum
{
    JET_BLACK, RED, PALE_GREY, YELLOW, ..., LAST_STR
} STR;
static const STRING strings[LAST_STR] =
{
    { /* JET_BLACK */
        15,
        {
            "Jet Black ",          /* English */
            "Rabenschwarz ", /* German */
            ...
        }
    },
    { /* RED */
        15,
        {
            "Red ",                /* English */
            "Rot ",                /* German */
            ...
        }
    },
    ...
};
```

Listing 3: Final string storage structure

I did all of this manually, but you could certainly develop a script to automate the process of creating the enumerated list in the header file (from the contents of the string file).

Having gone through this exercise, I realized that I could make a bit more use of enumerated lists to make my code more readable and more maintainable. When the string data structure was introduced, a manifest constant N\_LANG was used to specify the number of languages supported. A better approach is as follows (with the changes in red):

```
typedef enum
{ ENGLISH, FRENCH, GERMAN, SPANISH, LAST_LANGUAGE }
LANGUAGE;
```

Now, the STRING data structure is defined as:

```
typedef struct
{
    /*
    * Maximum length
```

```
*/  
int const len;  
  
/*  
 * Array of pointers to language-specific string  
 */  
char const * const text[LAST_LANGUAGE];  
  
} STRING;
```

This change may look minor, but it makes adding another language more intuitive.

So far, I haven't mentioned the utility of storing the maximum string length in the STRING data structure. The use of this field arises when ROM is not plentiful, such that it is necessary to store strings without padding out to the maximum allowed field width. In cases like this, one has to be careful to clear the entire field before writing the string. This may be accomplished by using the string len parameter. If you can afford to pad all strings out to the allowed field width, it is permissible to drop the len parameter from the data structure.

Well, that's my fourth attempt at producing an international product. After three disasters, I'm reasonably confident that this latest attempt will at least make it into the "not bad" category. If you have any refinements that you would care to share, please e-mail me. In the meantime, I'm going to ponder how to elegantly and robustly support languages such as Chinese, Arabic and Russian in embedded systems. If I manage to find a reasonable solution, I'll let you know.

---

This article was published in the February 2001 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. "Support Multiple Languages" Embedded Systems Programming, February 2001.



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

Nigel Jones ■

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 function 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_SCREEN);
/* 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 PFV_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 2
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 PfcPC_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 an INT as an argument 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 i); // Example function prototype

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

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

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

### Example 8

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 PfcPcC_PcI[] = {fna,fnb,fnc, ... fnz};

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

// Direct declaration
static const CHAR * const (* const pf[])(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 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 PFCPC_CPCI)(const INT * const);
static PVCPC_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 PFcPvC_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.