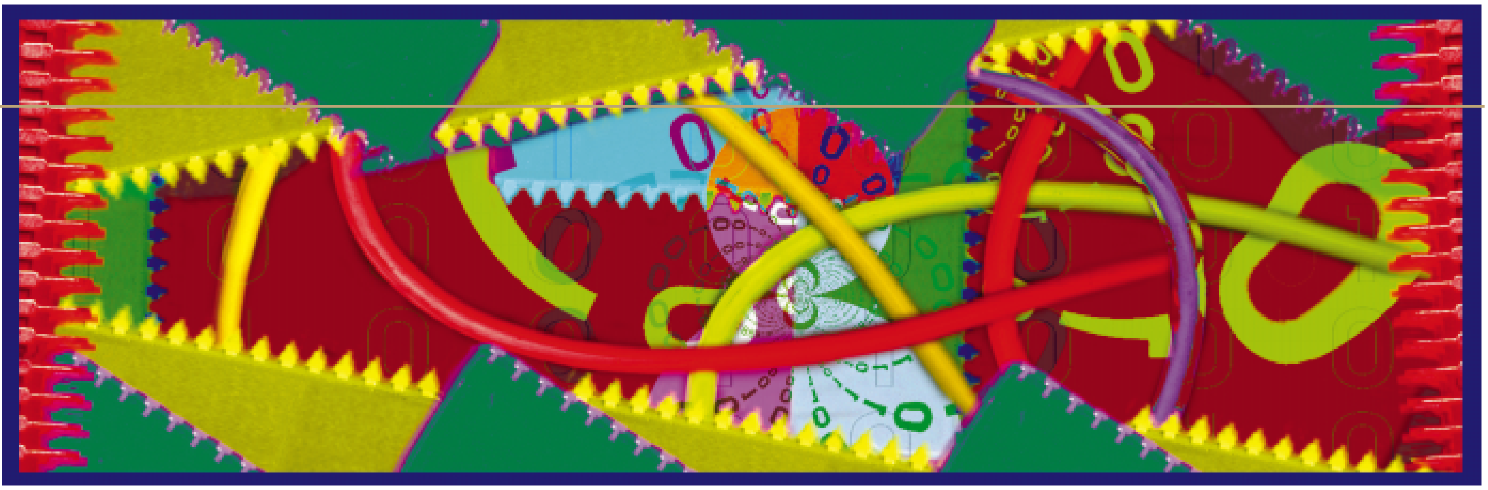
RETURN

NIGEL JONES

feature



Efficient C Code for Eight-Bit MCUs

The 8051, 68HC11, and PIC are popular MCUs, but they aren't necessarily easy to program. This article shows how the use of ANSI and compiler-specific constructs can help generate tighter code.

Getting the best possible performance out of an eight-bit microcontroller C compiler isn't always easy. This article concentrates mainly on those microcontrollers that were never designed to support high-level languages, such as the 8051 family, the 6800 family (including the 68HCll), and the PIC line of microcontrollers. Newer eight-bit machines such as the Philips 8051XA and the Atmel Atmega series were designed explicitly to support HLLs, and as such, may not need all the techniques I describe here.

My emphasis is not on algorithm design, nor does it depend on a specific microprocessor or compiler. Rather, I describe general techniques that are widely applicable. In many cases, these techniques work on larger machines, although you may decide that the trade-offs involved aren't worthwhile.

Before jumping into the meat of the article, let's briefly digress with a discussion of the philosophy involved. The microcontrollers I mentioned are popular for reasons of size, price, power consumption, peripheral mix, and so on. Notice that "ease of programming" is conspicuously missing from this list. Traditionally, these microcontrollers have been programmed in assembly language. In the last few years, many vendors have recognized the desire of users to increase their productivity, and have introduced C compilers for these machines—many of which are extremely good. However, it's important to remember that no matter how good the compiler, the underlying hardware has severe limitations. Thus, to write efficient C for these targets, it's essential that we be aware of what the compiler can do easily and what requires compiler heroics. In presenting these techniques, I have taken the attitude that I wish to solve a problem by programming a microcontroller, and that the C compiler is a tool, no different from an oscilloscope. In other words, C is a means to an end, and not an end in

To write efficient C for these targets, it is essential that we be aware of what the compiler can do easily and what requires compiler heroics.

itself. As a result, many of my comments will seem heretical to the purists out there.

ANSI C

The first step to writing a realistic C program for an eight-bit machine is to dispense with the concept of writing 100% ANSI code. This concession is necessary because I don't believe it's possible, or even desirable, to write 100% ANSI code for any embedded system, particularly for eight-bit systems. Some characteristics of eight-bit systems that prevent ANSI compliance are:

- Embedded systems interact with hardware. ANSI C provides extremely crude tools for addressing registers at fixed memory locations. Consequently, most compiler vendors offer language extensions to overcome these limitations
- All nontrivial systems use interrupts. ANSI C doesn't have a standard way of coding interrupt service routines
- ANSI C has various type promotion rules that are absolute performance killers to an eight-bit machine. Unless your system has abundant CPU cycles, you will quickly learn to defeat the ANSI promotion rules
- Many microcontrollers have multiple memory spaces, which have to be specified in order to correctly address the desired variable. Thus, variable declarations tend to be considerably more complex than on the typical PC application
- Many microcontrollers have no hardware support for a C stack. Consequently, some compiler vendors dispense with a stack-based architecture, in the process eliminating several key features of C

This is not to say that I advocate junking the entire ANSI standard. Indeed, some of the essential requirements of the standard, such as function prototyping, are invaluable. Rather, I take the view that one should use standard C as much as possible. However, when it interferes with solving the problem at hand, do not hesitate to bypass it. Does this interfere with making code portable and reusable? Absolutely. But portable, reusable code that doesn't get the job done isn't much use.

I've also noticed that every compiler has a switch that strictly enforces ANSI C and disables all compiler extensions. I suspect that this is done purely so that a vendor can claim ANSI compliance, even though this feature is practically useless. I have also observed that vendors who strongly emphasize their ANSI compliance often produce inferior code (perhaps because the compiler has a generic front end that is shared among multiple targets) when compared to vendors that emphasize their performance and language extensions.

Enough on the ANSI standard—let's address specific actions that can be taken to make your code run well on an eight-bit microcontroller. The most important, by far, is the choice of data types.

Data types

Knowledge of the size of the underlying data types, together with careful data type selection, is essential for writing efficient code on eight-bit machines. Furthermore, understanding how the compiler handles expressions involving your data types can make a considerable difference in your coding decisions. These topics are discussed in the following paragraphs.

Data type size

In the embedded world, knowing the underlying representation of the various data types is usually essential. I have seen many discussions on this topic, none of which has been particularly satisfactory or portable. My preferred solution is to include a file, `<types.h>`, an excerpt from which appears below:

```
#ifndef TYPES_H
#define TYPES_H
#include <limits.h>
/* Assign a built in data type to
   BOOLEAN. This is compiler
   specific */
#ifdef _C51_
typedef bit BOOLEAN
#define FALSE 0
#define TRUE 1
#else
typedef  enum  {FALSE=0,  TRUE=1}
   BOOLEAN;
#endif
/* Assign a built in data type to
  type CHAR. This is an eight-bit
  signed variable */
#if (SCHAR_MAX == 127)
typedef char CHAR;
#elif (SCHAR_MAX == 255)
/* Implies that by default chars
  are unsigned */
typedef signed char CHAR;
#else
/* No eight bit data types */
#error Warning! Intrinsic data type
  char is not eight bits!!
#endif
/* Rest of the file goes here */
#endif
```

The concept is quite simple. `Types.h` includes the ANSI-required file `limits.h`. It then explicitly tests each of the predefined data types for the smallest type that matches signed and unsigned one-, eight-, 16-, and 32-

NANCE PATERNOSTER

bit variables. The result is that my data type `UCHAR` is guaranteed to be an eight-bit unsigned variable, `INT` is guaranteed to be a 16-bit signed variable, and so forth. In this manner, the following data types are defined: `BOOLEAN`, `CHAR`, `UCHAR`, `INT`, `UINT`, `LONG`, and `ULONG`. Several points are worth making:

- The definition of the `BOOLEAN` data type is difficult. Many eight-bit machines directly support single-bit data types, and I wish to take advantage of this if possible. Unfortunately, since ANSI is silent on this topic, it's necessary to use compiler-specific compilation
- Some compilers define a `char` as an unsigned quantity, such that if a signed eight-bit variable is required, one has to use the unusual declaration `signed char`
- Note the use of the error function to force a compile error if I can't achieve my goal of having unambiguous definitions of `BOOLEAN`, `UCHAR`, `CHAR`, `UINT`, `INT`, `ULONG`, and `LONG`

In all of the following examples, the types `BOOLEAN`, `UCHAR`, and so on will be used to specify unambiguously the size of the variable being used.

Data type selection

There are two rules for data type selection on eight-bit machines:

- Use the smallest possible type to get the job done
- Use an unsigned type if possible

The reasons for this are simply that many eight-bit machines have no direct support for manipulating anything more complicated than an unsigned eight-bit variable. However, unlike large machines, eight-bitters often provide direct support for manipulation of bits. Thus, the fastest integer types to use on an eight-bit machine are `BOOLEAN` and `UCHAR`. Consider the typical C code:

On many compilers, declaration of an enumerated type forces the compiler to generate 16-bit signed code, which, as I've mentioned, is extremely inefficient.

```
int is_positive(int a)
{
  (a>=0) ? return(1) : return (0);
}
```

The better implementation is:

```
BOOLEAN is_positive(int a)
{
   (a>=0)  ?  return(TRUE)  :  return
(FALSE);
}
```

On an eight-bit machine we can get a large performance boost by using the `BOOLEAN` return type because the compiler need only return a bit (typically via the carry flag), vs. a 16-bit value stored in registers. The code is also more readable.

Let's take a look at a second example. Consider the following code fragment that is littered throughout most C programs:

```
int j;
for(j=0; j<10; j++)
  {
  :
  }
```

This fragment produces horribly inefficient code on an 8051. The correct way to code this for eight-bit machines is as follows:

```
UCHAR j;
for (j=0; j<10; j++)
  {
  :
  }
```

The result is a huge boost in performance because we are now using an eight-bit unsigned variable (that can be manipulated directly) vs. a signed 16-bit quantity that will typically be handled by a library call. Note also that no penalty exists for coding this way on most big machines (with the exception of some RISC processors. Furthermore, a strong case exists for doing this on *all* machines. Those of you who know Pascal are aware that when declaring an integer variable, it's possible, and normally desirable, to specify the allowable range that the integer can take on. For example:

```
type loopindex = 0..9;
var j loopindex;
```

Upon rereading the code later, you'll have additional information concerning the intended use of the variable. For our classical C code above, the variable `int j` may take on values of at least –32768 to +32767. For the case in which we have `uchar j`, we inform others that this variable is intended to have strictly positive values over a restricted range. Thus, this simple change manages to combine tighter code with improved maintainability—not a bad combination.

Enumerated types

The use of enumerated data types was a welcome addition to ANSI C. Unfortunately, the standard calls for the underlying data type of an enum to be an `int`. Thus, on many compilers, declaration of an enumerated type forces the compiler to generate 16-bit signed code, which, as I've mentioned, is extremely inefficient. This is unfortunate, especially as I have never seen an enumerated type list go over a few dozen elements, such that it could easily be fit in a `UCHAR`. To overcome this limitation, several options exist, none of which is palatable:

- Check your compiler documentation. It may allow you to specify via a command line switch that enumerated types be put into the smallest possible data type. The downside is, of course, compiler-dependant code

The integer promotion rules of ANSI C are probably the most heinous crime committed against those of us who labor in the eight-bit world.

- Accept the inefficiency as an acceptable trade-off for readability
- Dispense with enumerated types and resort to lists of manifest constants

Integer promotion rules

The integer promotion rules of ANSI C are probably the most heinous crime committed against those of us who labor in the eight-bit world. I have no doubt that the standard is quite detailed in this area. However, the two most important rules in practice are the following:

- Any expression involving integral types smaller than an `int` have all the variables automatically promoted to `int`
- Any function call that passes an integral type smaller than an `int` automatically promotes the variable to an `int`, if the function is not prototyped. (Yet another reason for using function prototyping)

The key word here is *automatically*. Unless you take explicit steps, the compiler is unlikely to do what you want. Consider the following code fragment:

```
CHAR a,b,res;
:
res = a+b;
```

The compiler will promote `a` and `b` to integers, perform a 16-bit addition, and then assign the lower eight bits of the result to `res`. Several ways around this problem exist. First, many compiler vendors have seen the light, and allow you to disable the ANSI automatic integer promotion rules. However, you're then stuck with compiler-dependant code.

Alternatively, you can resort to very clumsy casting, and hope that the compiler's optimizer works out what you really want to do. The extent of the casting required seems to vary among compiler vendors. As a result, I tend to go overboard:

```
res = (CHAR)((CHAR)a + (CHAR)b);
```

With complex expressions, the result can be hideous.

More integer promotion rules

A third integer promotion rule that is often overlooked concerns expressions that contain both signed and unsigned integers. In this case, signed integers are promoted to unsigned integers. Although this makes sense, it can present problems in our eight-bit environment, where the unsigned integer rules. For example:

```
void demo(void)
 {
 UINT a = 6;
 INT b = -20;

 (a+b > 6) ? puts("More than 6")
: puts("Less than or equal to 6");
 }
```

If you run this program, you may be surprised to find that the output is "More than 6." This problem is a very subtle one, and is even more difficult to detect when you use enumerated data types or other defined data types that evaluate to a signed integer data type. Using the result of a function call in an expression is also problematic.

The good news is that in the embedded world, the percentage of integral data types that *must* be signed is quite low, thus the potential number of expressions in which mixed types occur is also low. The time to be cautious is when reusing code that was written by someone who didn't believe in unsigned data types.

Floating-point types

Floating-point arithmetic is required in many applications. However, since we're normally dealing with real-world data whose representation rarely goes beyond 16 bits (a 20-bit `atod` on an eight-bit machine is rare), the requirements for double-precision arithmetic are tenuous, except in the strangest of circumstances. Again, the ANSI people have handicapped us by requiring that any floating-point expression be promoted to double before execution. Fortunately, a lot of compiler vendors have done the sensible thing, and simply defined doubles to be the same as floats, so that this promotion is benign. Be warned, however, that many reputable vendors have made a virtue out of providing a genuine double-precision data type. The result is that unless you take great care, you may end up computing values with ridiculous levels of precision, and paying the price computationally. If you're considering a compiler that offers double-precision math, study the documentation carefully to ensure that there is some way of disabling the automatic promotion. If there isn't, look for another compiler.

While we're on this topic, I'd like to air a pet peeve of mine. Years ago, before decent compiler support for eight-bit machines was available, I would code in assembly language using a bespoke floating-point library. This library was always implemented using three-byte floats, with a long float consuming four bytes. I found that this was more than adequate for the real world. I've yet to find a compiler vendor that offers this as an option. My guess is that the marketing people insisted on a true ANSI floating-point library, the real world be damned. As a result, I can calculate hyperbolic sines on my 68HC11, but I can't get the performance boost that comes from using just a three-byte float.

Having moaned about the ANSI-induced problems, let's turn to an area in which ANSI has helped a lot.

While the use of static functions is good structured programming practice, you may also be surprised to learn that static functions can result in smaller and/or faster code.

I'm referring to the key words `const` and `volatile`, which, together with `static`, allow the production of better code.

Key words

The three key words `static`, `volatile`, and `const` together allow one to write not only better code (in the sense of information hiding and so forth) but also tighter code.

Static variables

When applied to variables, `static` has two primary functions. The first and most common use is to declare a variable that doesn't disappear between successive invocations of a function. For example:

```
void func(void)
{
  static UCHAR state = 0;
  switch (state)
    {
    :
    }
}
```

In this case, the use of `static` is mandatory for the code to work.

The second use of `static` is to limit the scope of a variable. A variable that is declared `static` at the module level is accessible by all functions in the module, but by no one else. This is important because it allows us to gain all the performance benefits of global variables, while severely limiting the well-known problems of globals. As a result, if I have a data structure which must be accessed frequently by a number of functions, I'll put all of the functions into the same module and declare the structure `static`. Then all of the functions that need to can access the data without going through the overhead of an access function, while at the same time, code that has no business knowing about the data structure is prevented from accessing it. This technique is an admission that directly accessible variables are essential to gaining adequate performance on small machines.

A few other potential benefits can result from declaring module level variables `static` (as opposed to leaving them global). Static variables, by definition, may only be accessed by a specific set of functions. Consequently, the compiler and linker are able to make sensible choices concerning the placement of the variables in memory. For instance, with static variables, the compiler/linker may choose to place all of the static variables in a module in contiguous locations, thus increasing the chances of various optimizations, such as pointers being simply incremented or decremented instead of being reloaded. In contrast, global variables are often placed in memory locations that are designed to optimize the compiler's hashing algorithms, thus eliminating potential optimizations.

Static functions

A static function is only callable by other functions within its module. While the use of static functions is good structured programming practice, you may also be surprised to learn that static functions can result in smaller and/or faster code. This is possible because the compiler knows at compile time exactly what functions can call a given static function. Therefore, the relative memory locations of functions can be adjusted such that the static functions may be called using a short version of the call or jump instruction. For instance, the 8051 supports both an ACALL and an LCALL op code. ACALL is a two-byte instruction, and is limited to a 2K address block. `LCALL` is a three-byte instruction that can access the full 8051 address space. Thus, use of static functions gives the compiler the opportunity to use an ACALL where otherwise it might use an LCALL.

The potential improvements are even better, in which the compiler is smart enough to replace calls with jumps. For example:

```
void fa(void)
{
:
fb();
}

static void fb(void)
{
:
}
```

In this case, because function `fb()` is the last line of function `fa()`, the compiler can substitute a call with a jump. Since `fb()` is `static`, and the compiler knows its exact distance from `fa()`, the compiler can use the shortest jump instruction. For the Dallas DS80C320, this is an `SJMP` instruction (two bytes, three cycles) vs. an LCALL (three bytes, four cycles).

On a recent project of mine, rigorous application of the static modifier to functions resulted in about a 1% reduction in code size. When your EPROM is 95% full (the normal case), a 1% reduction is most welcome!

A final point concerning static variables and debugging: for reasons that I do not fully understand, with many in-circuit emulators that support source-level debug, static variables and/or automatic variables in static functions are not always accessible symbolically. As a result, I tend to use the following construct in my project-wide `include` file:

```
#ifndef NDEBUG
#define STATIC
#else
#define STATIC static
#endif
```

I then use `STATIC` instead of `static` to define static variables, so that while in debug mode, I can guarantee symbolic access to the variables.

Volatile variables

A volatile variable is one whose value may be changed outside the normal program flow. In embedded systems, the two main ways that this can happen is either via an interrupt service routine, or as a consequence of hardware action (for instance, a serial port status register updates as a result of a character being received via the serial port). Most programmers are aware that the compiler will not attempt to optimize a volatile register, but rather will reload it every time. The case to watch out for is when compiler vendors offer extensions for accessing absolute memory locations, such as hardware registers. Sometimes these extensions have either an implicit or an explicit declaration of volatility and sometimes they don't. The point is to fully understand what the compiler is doing. If you do not, you may end up accessing a volatile variable when you don't want to and vice versa. For example, the popular 8051 compiler from Keil offers two ways of accessing a specific memory location. The first uses a language extension, `_at_`, to specify where a variable should be located. The second method uses a macro such as `XBYTE[]` to dereference a pointer. The "volatility" of these two is different. For example:

```
UCHAR status_register  _at_ 0xE000;
```

This method is simply a much more convenient way of accessing a specific memory location. However, `volatile` is not implied here. Thus, the following code is unlikely to work:

```
while(status_register)
  ; /* Wait for status register to
clear */
```

Instead, one needs to use the following declaration:

```
volatile  UCHAR  status_register
_at_ 0xE000;
```

The second method that Keil offers is the use of macros, such as the `XBYTE` macro, as in:

```
status_register = XBYTE[0xE000];
```

Here, however, examination of the `XBYTE` macro shows that `volatile` is assumed:

```
#define  XBYTE  ((unsigned  char
volatile xdata*)  0)
```

(The `xdata` is a memory space qualifier, which isn't relevant to the discussion here and may be ignored.)

Thus, the code:

```
while(status_register)
  ; /* Wait for status register to
clear */
```

will work as you would expect in this case. However, in the case in which you wish to access a variable at a specific location that is not volatile, the use of the `XBYTE` macro is potentially inefficient.

Const variables

The keyword `const`, the most badly named keyword in the C language, does not mean *constant*! Rather, it means "read only." In embedded systems, there is a huge difference, which will become clear.

Const variables vs. manifest constants

Many texts recommend that instead of using manifest constants, one should use a `const` variable. For instance:

```
const UCHAR nos_atod_channels = 8;
```

instead of

```
#define NOS_ATOD_CHANNELS 8
```

The rationale for this approach is that inside a debugger, you can examine a `const` variable (since it should appear in the symbol table), whereas a manifest constant isn't accessible. Unfortunately, on many eight-bit machines you'll pay a significant price for this benefit. The two main costs are:

- The compiler creates a genuine variable in RAM to hold the variable. On RAM-limited systems, this can be a significant penalty
- Some compilers, recognizing that the variable is `const`, will store the variable in ROM. However, the variable is still treated as a variable and is accessed as such, typically using some form of indexed addressing. Compared to immediate addressing, this method is normally much slower

I recommend that you eschew the use of `const` variables on eight-bit machines, except in the following certain circumstances.

Declaring function parameters `const` whenever possible not only makes for better, safer code, but also has the potential for generating tighter code.

We now come to an esoteric topic. Can a variable be both `const` and `volatile`, and if so, what does that mean and how might you use it?

Const function parameters

Declaring function parameters `const` whenever possible not only makes for better, safer code, but also has the potential for generating tighter code. This is best illustrated by an example:

```
void output_string(CHAR *cp)
{
  while (*cp)
    putchar(*cp++);
}

void demo(void)
{
char *str = "Hello, world";

output_string(str);

if ('H' == str[0]) {
  some_function();
  }
}
```

In this case, there is no guarantee that `output_string()` will not modify our original string, `str`. As a result, the compiler is forced to perform the test in `demo()`. If instead, `output_string` is correctly declared as follows:

```
void output_string(const char *cp)
{
  while (*cp)
    putchar(*cp++);
}
```

then the compiler knows that `output_string()` cannot modify the original string `str`, and as a result it can dispense with the test and invoke `some_function()` unconditionally. Thus, I strongly recommend liberal use of the `const` modifier on function parameters.

Const volatile variables

We now come to an esoteric topic. Can a variable be both `const` and `volatile`, and if so, what does that mean and how might you use it? The answer is, of course, yes (why else would it have been asked?), and it should be used on any memory location that can change unexpectedly (hence the need for the `volatile` qualifier) and that is read-only (hence the `const`). The most obvious example of this is a hardware status register. Thus, returning to the `status_register` example above, a better declaration for our status register is:

```
const volatile UCHAR status_reg-
  ister _at_ 0xE000;
```

Typed data pointers

We now come to another area in which a major trade-off exists between writing portable code and writing efficient code—namely the use of *typed data pointers*, which are pointers that are constrained in some way with respect to the type and/or size of memory that they can access. For example, those of you who have programmed the x86 architecture are undoubtedly familiar with the concept of using the `__near` and `__far` modifiers on pointers. These are examples of typed data pointers. Often the modifier is implied, based on the memory model being used. Sometimes the modifier is mandatory, such as in the prototype of an interrupt handler:

```
void __interrupt __far
  cntr_int7();
```

The requirement for the near and far modifiers comes about from the segmented x86 architecture. In the embedded eight-bit world, the situation is often far more complex. Microcontrollers typically require typed data pointers because they offer a number of disparate memory spaces, each of which may require the use of different addressing modes. The worst offender is the 8051 family, with at least five different memory spaces. However, even the 68HC11 has at least two different memory spaces (zero page and everything else), together with the EEPROM, pointers to which typically require an address space modifier.

The most obvious characteristic of typed data pointers is their inherent lack of portability. They also tend to lead to some horrific data declarations. For example, consider the following declaration from the Whitesmiths 68HC11 compiler:

```
@dir INT * @dir
zpage_ptr_to_zero_page;
```

This declares a pointer to an `INT`. However, both the pointer and its object reside in the zero page (as indicated by the Whitesmith extension, `@dir`). If you were to add a `const` qualifier or two, such as:

```
@dir const INT * @dir const con-
stant_zpage_ptr_to_constant_zero_p
age_data;
```

then the declarations can quickly become quite intimidating. Consequently, you may be tempted to simply ignore the use of typed pointers. Indeed, coding an application on a 68HC11 without ever using a typed data pointer is quite possible. However, by doing so the application's performance will take an enormous hit because the zero page offers considerably faster access than the rest of memory.

This area is so critical to performance that all hope of portability is lost. For example, consider two leading 8051 compiler vendors, Keil and Tasking. Keil supports a three-byte generic pointer that may be used to point to any of the 8051 address spaces, together with typed data pointers that are strictly limited to a specific data space. Keil strongly recommends the use of typed data pointers, but doesn't require it. By contrast, Tasking

The most obvious characteristic of typed data pointers is their inherent lack of portability. They also tend to lead to some horrific data declarations.

takes the attitude that generic pointers are so horribly inefficient that it mandates the use of typed pointers (an argument to which I am extremely sympathetic).

To get a feel for the magnitude of the difference, consider the following code, intended for use on an 8051:

```
void main(void)
{
UCHAR array[16];  /* array is in
  the data space by default */
UCHAR data *ptr = array; /* Note
  use of data qualifier */
UCHAR i;

  for(i=0; i<16; i++)
    *ptr++ = i;
}
```

Using a generic pointer, this code requires 571 cycles and 88 bytes. Using a typed data pointer, it needs just 196 cycles and 52 bytes. (The memory sizes include the startup code, and the execution times are just those for executing `main()`).

With these sorts of performance increases, I recommend always using explicitly typed pointers, and paying the price in loss of portability and readability.

Use of assert

The `assert()` macro is commonly used on PC platforms, but almost never used on small embedded systems. There are several reasons for this:

- Many reputable compiler vendors don't bother to supply an `assert` macro
- Vendors that do supply the macro often provide it in an almost useless form
- Most embedded systems don't support a `stderr` to which the error may be printed

These limitations notwithstanding, it's possible to gain the benefits of the `assert()` macro on even the smallest systems if you're prepared to take a pragmatic approach.

Before I discuss possible implementations, mentioning why `assert()` is important (even in embedded systems) is worthwhile. Over the years, I've built up a library of drivers to various pieces of hardware such as LCDs, ADCs, and so on. These drivers typically require various parameters to be passed to them. For example, an LCD driver that displays a text string on a panel would expect the row, the column, a pointer to the string, and perhaps an attribute parameter. When writing the driver, it is obviously important that the passed parameters are correct. One way of ensuring this is to include code such as this:

```
void Lcd_Write_Str(UCHAR row,
  UCHAR column, CHAR *str, UCHAR
  attr)
{
  row &= MAX_ROW;
  column &= MAX_COLUMN;
  attr &= ALLOWABLE_ATTRIBUTES;

  if (NULL == str)
    return;

  /* The real work of the driver
  goes here */
}
```

This code clips the parameters to allowable ranges, checks for a null pointer assignment, and so on. However, on a functioning system, executing this code every time the driver is invoked is extremely costly. But if the code is discarded, reuse of the driver in another project becomes a lot more difficult because errors in the driver invocation are tougher to detect.

The preferred solution is the liberal use of an assert macro. For example:

```
void Lcd_Write_Str(UCHAR row,
  UCHAR column, CHAR *str, UCHAR
  attr)
{
  assert (row < MAX_ROW);
  assert (column < MAX_COLUMN);
  assert (attr <
    ALLOWABLE_ATTRIBUTES);
  assert (str != NULL);

  /* The real work of the driver
goes here */
}
```

This is a practical approach if you're prepared to redefine the assert macro. The level of resources in your system will control the sophistication of this macro, as shown in the examples below.

Assert #1

This example assumes that you have no spare RAM, no spare port pins, and virtually no ROM to spare. In this case, assert.h becomes:

```
#ifndef assert_h
#define assert_h
#ifndef NDEBUG

#define assert(expr) \
  if (expr) {\
    while (1);\
  }
#else
#define assert(expr)
#endif
#endif
```

Here, if the assertion fails, we simply enter an infinite loop. The only utility of this case is that, assuming you're running a debug session on an ICE, you will eventually notice that the system is no longer running. In which case, breaking the emulator and examining the program counter will give you a good indication of which assertion failed. As a possible refinement, if your system is interrupt-driven, inserting a "disable all interrupts" command prior to the `while(1)` may

Recursion is a wonderful technique that solves certain problems in an elegant manner. It has no place on an eight-bit microcontroller.

be necessary, just to ensure that the system's failure is obvious.

Assert #2

This case is the same as assert #1, except that in #2 you have a spare port pin on the microcontroller to which an error LED is attached. This LED is lit if an error occurs, thus giving you instant feedback that an assertion has failed. Assert.h now becomes:

```
#ifndef assert_h
#define assert_h
#define ERROR_LED_ON() /* Put
  expression for turning LED on
  here */
#define INTERRUPTS_OFF() /* Put
  expression for interrupts off
  here */

#ifndef NDEBUG

#define assert(expr) \
  if (expr) {\
    ERROR_LED_ON();\
    INTERRUPTS_OFF();\
    while (1);\
  }
#else
#define assert(expr)
#endif
#endif
```

Assert #3

This example builds on assert #2. But in this case, we have sufficient RAM to define an error message buffer, into which the assert macro can `sprintf()` the exact failure. While debugging on an ICE, if a permanent watch point is associated with this buffer, then breaking the ICE will give you instant information on where the failure occurred. Assert.h for this case becomes:

```
#ifndef assert_h
#define assert_h
#define ERROR_LED_ON()   /* Put
  expression for turning LED on
  here */
#define INTERRUPTS_OFF()/* Put
  expression for interrupts off
  here */

#ifndef NDEBUG
extern char error_buf[80];

#define assert(expr) \
  if (expr) {\
    ERROR_LED_ON();\
    INTERRUPTS_OFF();\
    sprintf(error_buf,"Assert
    failed: " #expr " (file %s
    line %d)\n",
__FILE__, (int) __LINE__ );\
    while (1);\
    }
#else
#define assert(expr)
#endif
#endif
```

Obviously, this requires that you define `error_buffer[80]` somewhere else in your code.

I don't expect that these three examples will cover everyone's needs. Rather, I hope they give you some ideas on how to create your own assert macros to get the maximum debugging information within the constraints of your embedded system.

Heretical comments

So far, all of my suggestions have been about actively doing things to improve the code quality. Now, let's address those areas of the C language that should be avoided, except in highly unusual circumstances. For some of you, the suggestions that follow will border on heresy.

Recursion

Recursion is a wonderful technique that solves certain problems in an elegant manner. It has no place on an eight-bit microcontroller. The reasons for this are quite simple:

- Recursion relies on a stack-based approach to passing variables. Many small machines have no hardware support for a stack. Consequently, either the compiler will simply refuse to support reentrancy, or else it will resort to a software stack in order to solve the problem, resulting in dreadful code quality
- Recursion relies on a "virtual stack" that purportedly has no real memory constraints. How many small machines can realistically support virtual memory?

If you find yourself using recursion on a small machine, I respectfully suggest that you are either a) doing something really weird, or b) you don't understand the sum total of the constraints with which you're working. If it is the former, then please contact me, as I will be fascinated to see what you are doing.

Variable length argument lists

You should avoid variable length argument lists because they too rely on a stack-based approach to passing variables. What about `sprintf()` and its cousins, you all cry? Well, if possible, you should consider avoiding the use of these library functions. The reasons for this are as follows:

- If you use `sprintf()`, take a look at the linker output and see how much library code it pulls in. On one of my compilers, `sprintf()`, without floating-point support, consumes about 1K. If you're using a masked micro with a code space of 8K, this penalty is huge
- On some compilers, use of `sprintf()` implies the use of a floating-point library, even if you never use the library. Consequently, the code penalty quickly becomes enormous
- If the compiler doesn't support a stack, but rather passes variables in registers or fixed memory loca-

If you're using recursion on a small machine, I respectfully suggest that you are either a) doing something really weird, or b) you don't understand the sum total of the constraints with which you're working.

tions, then use of variable length argument functions forces the compiler to reserve a healthy block of memory simply to provide space for variables that you may decide to use. For instance, if your compiler vendor assumes that the maximum number of arguments you can pass is 10, then the compiler will reserve 40 bytes (assuming four bytes per longest intrinsic data type)

Fortunately, many vendors are aware of these issues and have taken steps to mitigate the effects of using `sprintf()`. Notwithstanding these actions, taking a close look at your code is still worthwhile. For instance, writing my own `wrstr()` and `wrint()` functions (to ouput strings and `ints` respectively) generated half the code of using `sprintf`. Thus, if all you need to format are strings and base 10 integers, then the roll-your-own approach is beneficial (while still being portable).

Dynamic memory allocation

When you're programming an application for a PC, using dynamic memory allocation makes sense. The characteristics of PCs that permit and/or require dynamic memory allocation include:

- When writing an application, you may not know how much memory will be available. Dynamic allocation provides a way of gracefully handling this problem
- The PC has an operating system, which provides memory allocation services
- The PC has a user interface, such that if an application runs out of memory, it can at least tell the user and attempt a relatively graceful shutdown

In contrast, small embedded systems typically have none of these characteristics. Therefore, I think that the use of dynamic memory allocation on these targets is silly. First, the amount of memory available is fixed, and is typically known at design time. Thus static allocation of all the required and/or available memory may be done at compile time.

Second, the execution time overhead of `malloc()`, `free()`, and so on is not only quite high, but also variable, depending on the degree of memory fragmentation.

Third, use of `malloc()`, `free()`, and so on consumes valuable EPROM space. And lastly, dynamic memory allocation is fraught with danger (witness the recent series from P.J. Plauger on garbage collection in the January 1998, March 1998, and April 1998 issues of *ESP*).

Consequently, I strongly recommend that you not use dynamic memory allocation on small systems.

Final thoughts

I have attempted to illustrate how judicious use of both ANSI constructs and compiler-specific constructs can help generate tighter code on small microcontrollers. Often, though, these improvements come at the expense of portability and/or readability. If you are in the fortunate position of being able to use less efficient code, then you can ignore these suggestions. If, however, you are severely resource-constrained, then give a few of these techniques a try. I think you'll be pleasantly surprised. **esp**

Nigel Jones received a degree in engineering from Brunel University, London. He has worked in the industrial control field, both in the U.K. and the U.S. Presently he is working as a consultant, with the majority of his work concentrated on underwater instrumentation. He can be reached at NAJones@compuserve.com.