SERVING THE EMBEDDED COMMUNITY FOR 20 YEARS 1988 20 YEARS 2008

BY ROBIN KNOKE

Debugging embedded C

Has debugging embedded C changed in 20 years? You betcha. But the process will never change: stabilize, isolate, correct, and retest. Here's an article from the 1988 premiere issue of *Embedded Systems Programming,* with some comments from the author, Robin Knoke.

Twenty years ago I wrote this article, and I still sometimes get it out and read it for a good laugh. As you know, much has changed since those days, and debugging embedded C is, in many respects, easier than it was. The basic steps in troubleshooting are still the same, however:

1. Make the bug repeatable
2. Isolate the problem.
3. Make corrections.
4. Retest.

It's interesting to note that in my recent experience, programmers still make some of those same mistakes. The tools are better, but the programs have become so much larger that I think we have a net loss, overall.

It has become increasingly more important that we organize our project, breaking it down into manageable sizes and thoroughly testing each piece. Serious programmers will adopt a naming convention for functions and variables and use header files and function prototyping. Choose a coding style for indenting and braces—and stick with it. These disciplines help tremendously during integration, especially when working in a team.

Even with new tools at our disposal, it can be very challenging to eliminate all of the bugs from a large program. It's only through careful organization, lots of testing, and strict attention to detail that we can achieve success. *—Robin Knoke, 12/07/08*

Debugging is one of a series of steps necessary to produce quality software. It consumes much of a programmer's time, yet is one of the least discussed and studied tasks in software development. The process of debugging, as described by Robert Ward in his book *Debugging C* (Que Corp., 1986), involves four phases: testing, stabilization, localization, and correction.

Testing exercises the capabilities of a program by stimulating it with a wide range of input values. First, the program is tested under normal conditions. If it appears to work, its handling of special cases and boundary conditions is tested. Tests should be carefully engineered to force execution of all program branches and thus ensure that every decision node is executed correctly. Any peculiar performance by the program during testing is considered a potential bug and should be investigated.

Stabilization, the second phase, is an attempt to control conditions to the extent that a specific bug can be generated at will. Usually a given set of test conditions will cause a bug to appear, and the bug will remain even when statements are added in the source code. As we'll see, however, certain classes of bugs typical in embedded C programs are difficult to stabilize; any change in the source code or linking process can significantly alter the bugs' behavior or even make them appear to go away.

Lexical and syntactic bugs are identified by the compiler at compile time; intent and execution bugs are identified by testing the program at run time.

In the localization stage, the programmer moves in for the kill. Localization involves narrowing the range of possibilities until the bug can be isolated to a specific variable or segment of code. There are three general approaches to this problem.

One approach is to construct a hypothesis to explain how such data might be created, then modify the experiment to test the validity of the hypothesis. This process of localization employs the scientific method of problem solving, requiring skills quite different from those needed to generate the code in the first place. Modifying the experiment itself will also cause some bugs to behave differently.

Another way to localize a bug is to single-step through the suspect code, watching carefully for the first sign of abnormal behavior. Since the programmer knows what's supposed to happen, the problem can often be pinpointed the first time through the program. The problems with this technique are that it tends to be tedious when the program contains loops or complex structures and that quite often the bug won't manifest itself during single-step execution.

Bugs can also be localized by examining a trace history of the executed code. Microprocessor emulators can be used to capture a trace of the program as it executes, and hardware breakpoints can be used to stop execution where desired. The trace history is then used to reconstruct what happened when the bug occurred.

Correction is the final step. After a bug is located, it must be eradicated. Often, correction is straightforward; sometimes, however, a bug reflects a conceptual design flaw. In any event, after the bug is corrected, the process starts over from the testing phase.

Debugging a program running under the supervision of an operating system can be quite different from debugging a program in an embedded system. The primary distinction is that the available tools are different. An operating system environment may support native debuggers, where an embedded system may not. Each system has its advantages. In the native environment, the compiler, source code, debugger, and target program are all together in one place. Cross-debugging—debugging programs in a separate target system—requires a monitor or an emulator. The code must be moved back and forth between the development system and the target. The advantage is that emulators provide hardware specifically designed to facilitate the debugging process.

The following process is typical of a software development effort once the specification for an embedded systems program has been determined:

1. Design the program.
2. Code and edit the program (during this step we may learn how to do step 1.)
3. Compile the program. If compile-time errors occur, localize to find the errors, then return to step 2.
4. Link the program. If link-time errors occur, modify the link commands and relink. If necessary, return to step 2.
5. Test the program. If run-time errors occur, stabilize and localize the bugs, then return to step 2. If necessary, go back to step 1.

To evaluate debugging tools, we need to understand the sources of potential bugs. In *Karel the Robot* (John Wiley & Sons, 1981), Richard Pattis cites four classes of bugs: lexical, syntactic, intent, and execution. Lexical and syntactic bugs are identified by the compiler at compile time; intent and execution bugs are identified by testing the program at run time.

COMPILE-TIME ERRORS

The compiler makes several passes through the code during compilation. The actual number of passes depends on the compiler, but most compilers make three or four. Bugs are usually discovered only in the first two passes. The first pass, made by the preprocessor, expands macros and reads in-eluded header files or other source files. In the next pass, the parser and lexical analyzer attempt to understand and produce cede from the source statements. Most of the error messages are generated during this second pass.

The following are simple examples of compile-time errors:

- Invalid preprocessor directives (`#page: 7`)
- Illegal operator use (`&a=123;`)
- Illegal symbol or identifier name (`byte j;`)
- Illegal punctuation or character (`j=0:`)
- Illegal language grammar (`if j==0 then j=3;`)
- Incompatible type operation (`*r = r;`)
- Invalid symbol or number (`int 23skidoo;`)

For the most part, these bugs result from types or errors made by a programmer still learning C and are classified as syntactical bugs. Stabilizing these bugs isn't an issue since the bug recurs each time the file is compiled.

Localization of a compile-time error is usually fairly easy, although occasionally a bug (such as an open comment) may require a little work to localize. Once the bug is localized, it's usually simple to correct.

LINK-TIME ERRORS

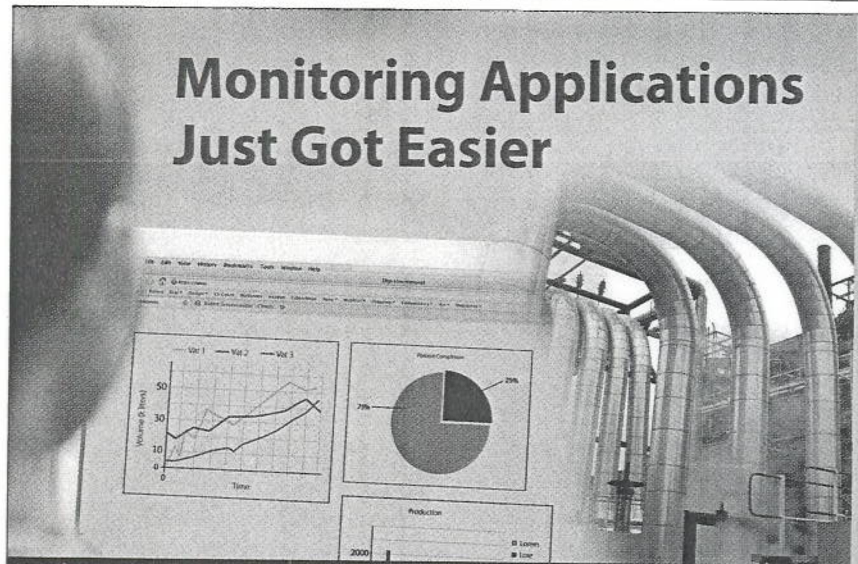
The linker's job is to connect other (hopefully tested) modules to the program and build an executable entity. It's possible that one or more of the modules linked to the target program will be either test routines capable of exercising the target program or stub modules simulating a section of code yet to be written.

If link-time errors are detected, most likely the linker couldn't find all the parts or libraries required to build the final program or couldn't understand the object modules representing the program due to incompatible format. (These problems are usually also reported by the compiler.) In either case, the error is in the commands to the linker or the format of the object module, not bugs in the program. Only rarely, such as when a library function name is misspelled, can a bug cause link errors.

The simplest intent bug is the one-liner, a correct statement but for misplaced punctuation or an incorrect assumption about operator precedence.

RUN-TIME ERRORS

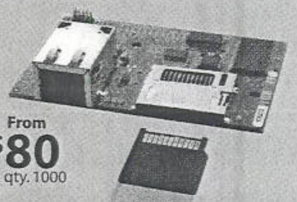
At this point, the programmer has suc-



Monitoring Applications Just Got Easier

Smarter Embedded Design

The RCM4300 RabbitCore® module gives you more processing power and memory for your demanding embedded applications that require web interfaces, graphics or encryption. Data logging applications will also take advantage of the miniSD™ memory cards with up to 1GB of storage. It's easy to add the RCM4300 – or any RabbitCore module – to your pin-compatible design.



From $80 qty. 1000

RCM4300 RabbitCore®

- Rabbit® 4000 @ 58.98 MHz
- 10/100Base-T Ethernet
- Up to 1GB of miniSD™
- 8 channel/12-bit analog

Get Your Kit Today for $299

Kits include Dynamic C® software, RCM4300 RabbitCore, 512MB miniSD™ card, development board, cables and accessories.

www.rabbitdevkits.com

RABBIT™

2900 Spafford St. Davis, CA 95618
Tel **530.757.8400**

08028

Listing 1 Code containing two potential boundary bugs.

```
/* Read n characters from stdin to buffer.
** Return EOF if end of file, otherwise return OK.
*/
int getinput(n,buffer)
int n;
char *buffer;
{
 int i, c;
 for (i=0; i<n; 1++) {
  c = fgetc(stdin);
  if (c==EOF)
    return(EOF);
  *buffer++ = c;
 } return(OK);
}
```

ceeded in getting the program to compile. This accomplishment doesn't mean that the program is free of typos or incorrectly formed statements; it simply means the compiler didn't detect them.

Now we come to the difficult bugs. Run-time bugs that exist without catastrophic results—the program will run but does the wrong things—are intent errors; those that cause the program to terminate abnormally are execution errors.

An intent error occurs whenever a program runs to normal completion but produces incorrect results. Examples of intent errors are one-liner, typematch, boundary, macro, and design bugs.

The simplest intent error is the one-liner, a syntactically correct statement that has an error in it. These errors are usually caused by an incorrect assumption regarding operator precedence, an incorrect choice of operator, or misplaced or missing punctuation. These errors, which I call "awshitical" bugs (based on mutterings from programmers who have spent considerable time staring right at the erroneous statement without seeing the bug), are a subclass of intent bugs. Here are a few:

```
if (i=1) {...}
```

Global variables are like salt: they should be used sparingly lest they spoil the stew. They often cause problems that surface during integration.

Dang, we wanted to compare `i`, not assign to it. The `if` condition will always be true.

```
if (i==1); {...}
```

Oops, we put a semicolon after the `if` statement. The `if` has no effect on the body of statements that follow.

```
while (c = getchar() != 0)
{...}
```

We forgot operator precedence; `c` will be 1 or 0. What we meant to write is `((c=getchar()) != 0)`.

Boundary bugs show up when test inputs are designed to test the boundary (or beyondboundary) conditions of the program. When dealing with arrays, it's easy to create a boundary bug by forgetting that the 10th element in a 10-element array is actually at `array [9]`. It's not uncommon for even the most seasoned programmer to occasionally generate invalid array indexes within `while` and `for` loops, especially if the loops are at all complex. Out-of-bounds array indexing can also cause viral bugs (described later).

Boundary bugs don't necessarily involve arrays. Any variable has a limited range of values, so tests at (and, if allowed, beyond) these limits should be run. Listing 1 contains two potential boundary bugs.

On machines that implement 16-bit ints, requesting more than 32,767 bytes will produce undesired results due to the signed comparison `i < n`. Also, if zero bytes are requested, the function can never return an EOF, even if no characters are available. This exception brings up an interesting point: if the programmer can guarantee that the calling function will never ask for zero characters, then this boundary need not be tested. Indeed, by definition the zero-length buffer bug doesn't exist.

A more complex form of bug is the type mismatch, or typematch, bug. It occurs when the programmer attempts to pass arguments to a function but the called function expects arguments of a different type. Although some of the newer compilers will catch this and some lint utilities are designed to look for precisely this type of error, typematch bugs still seem to crop up now and then. Here are two examples:

```
double nbr = 5.0;
int x;
sscanf("123","%d",x);
printf("%f",nbr);
```

In the third line, x should be &x; in the fourth line, the argument type is in-

Listing 2 Expansions of this macro will cause intent errors.

```
#define is_ascii_digit(x) ((x>'0'&&x<='9')?1:0)
func(nbr)
int nbr;
{
 if (is_ascii_digit(++nbr))
  printf("%c ",nbr);
 if (is_ascii_digit(nbr&0x7f))
  printf("%c ",nbr&0x7f);
}
```

correct. The function argument prototyping in newer compilers may catch these errors for library calls, but programmers can call their own functions and may not have included argument prototypes in their header files.

Macro bugs are errors that are inadvertently caused and camouflaged by macro expansion. When the preprocessor expands macros, it substitutes the macro definition anywhere the macro name appears in the body of the program. The programmer must be aware of what the code will actually look like after the macro expansion. If the macro contains parameters or other macros, there are even more things to consider. In Listing 2, both expansions of the macro will cause intent errors.

In the first expansion, `nbr` is incremented twice; in the second, the macro expands to `((nbr&0x7f > '0'&&nbr&0x7f<= '9') ?1:0)` and doesn't perform at all as might be expected. (My compiler concludes that the statement is always false and optimizes it to a jump instruction.)

Listing 3 An alphabetical sorting function containing design errors.

```
/* An ASCII collating function -
** return YES if s1 to follow s2, else return NO.
*/
alphasort(s1, s2)
char *s1, *s2;
{
 while (*s1!=0 && *s2!=0)
 {
  if (*s1 < *s2) return(NO);
  if (*s1 > *s2) return(YES);
  /* strings are equal so far */
  s1++; /* try next char in string */
  s2++;
 }
/* we reached the end of one string */
if (*s1) return(YES);
return(NO);
```

Intent bugs characterized by flaws in the design approach are called design bugs and result from incomplete comprehension of the problem. Some problems are so complex that it's hard to comprehend the entire problem at once. Sometimes the insight needed to solve the problem comes from trying to program it. Design bugs are often the result of some simple oversight by the programmer. Computers are more exact than we humans and sometimes embar-

Is your homegrown database embedded system performance?

Let an in-memory database system open the gates!

Homegrown databases don't scale and are slow, and that's b-a-a-a-a-d for embedded systems.

McObject's *eXtreme*DB™ in-memory database system (IMDS) delivers stunning performance, along with High Availability, an intuitive and type-safe API, and other critical features.

Count on *eXtreme*DB for embedded systems software that won't make you sheepish.

Download benchmark white papers and free trial *eXtreme*DB software at **www.mcobject.com**.

mcobject® precision data management
www.mcobject.com
Phone: +1-425-888-8505
Fax: +1-425-888-8508

Can CMX Really Put TCP/IP On My Little Ole 8-bit Chip?

Yes, Ma'am,

CMX has been doing amazing things with RTOSes and TCP/IP stacks for many years now. If you haven't visited us in a while, you are missing a lot of cool, new technology that is economical, royalty free, and comes with source code.

It might just put the sparkle back in your eyes!

CMX RTOSes and TCP/IP Stacks Support Most 8-, 16-, 32-bit Processors and DSP's.

CMX SYSTEMS

12276 San Jose Blvd
Jacksonville, FL 32223
Ph: 904.880.1840
Fax: 904.880.1632
email: cmx@cmx.com
www.cmx.com

Listing 4 Code containing an optimizer bug caused by memory-mapped I/O.

```
int sendack()
{
 /* address of device */
 extern char *data_port;
 while(*data_port == 0)
  /* wait here for 'ready' status */
  ;
 /* send 'ACK' and sequence */
 *data_port = 'A';
 *data_port = 'C';
 *data_port = 'K';
 /* return status to caller */
 return((int)(*data_port));
}
```

rass us with their explicit logic.

In the alphabetical sorting function in Listing 3, tests were run using all upper-case or all lowercase strings and the function worked flawlessly. When uppercase strings were compared with the lowercase the function claimed the uppercase string should come first, regardless of the characters. This wasn't expected.

Design bugs like these can be local to a function or result from the interface of two or more functions. As design errors span a larger scope, they are considered to be integration bugs.

Programs that terminate abnormally contain execution errors. These bugs may be detected by run-time type checks, bound checks, or hardware-fault detection mechanisms in native environments, but in an embedded application they may simply cause the system to crash. Examples of execution bugs are division by zero, running a program with link-time errors, incorrectly implemented interrupts or assembly code, out-of-bounds arrays, and assignment to an invalid (uninitialized) pointer.

Simple execution errors such as division by zero can be easy to stabilize, provided the zero is a direct result of a controlled test case. Otherwise, they can be hard to localize. For example, some processors issue an interrupt when an illegal operation is attempted. If this interrupt wasn't expected, the programmer may not have set a valid vector in the interrupt vector table. In this case, the control flow may go "into the weeds" and the original cause may not be readily apparent.

Many execution bugs are the result of an errant store through an incorrectly initialized pointer or at an out-of-bounds array index. These bugs belong to a very nasty class called viral bugs and can be extremely hard to stabilize and localize. They crop up in C programs because of the unrestricted run-time use of pointers, array indexing, and casting. The effect of a viral bug—corrupted code, data, or stack—is usually not apparent until several (million) instructions later. Even then the infected data might not prove catastrophic but may cause something else to become infected. At any rate, when the bug eventually surfaces, the symptom usually has nothing to do with the original bug.

Ordinarily, uninitialized pointers are more insidious than out-of-bounds arrays. An out-ofbounds array reference usually attacks a stack frame or data area adjacent to the location of the array. An uninitialized pointer can attack anywhere and is very often inconsiderate as to whether it attacks code or data. Furthermore, the pointer will probably contain a different initial value each time the program is run, causing an entirely different symptom each time.

SUBTLER BUGS

In addition to compile-time, link-time, and run-time errors, bugs resulting from integration, portability, and compiler errors may occur.

Integration bugs form a huge class of programming errors. These errors manifest themselves when two or more modules are combined to form a program. The bugs may not cause an error when the modules are tested separately but may show up as the system becomes integrated into one complex program. During integration, function return-value typematch bugs can become apparent. A function may return an error status if the data it processed is invalid. If the caller fails to check the error status, it may inadvertently continue processing with bogus data.

The opposite may also occur: a function that's supposed to return a value may instead contain a void return. In this case, the returned value is undefined and may appear to work during initial testing. Once integrated into a program, though, it creates an unexpected bug.

Global variables often cause problems that surface during integration. These variables are like salt: they should be used sparingly lest they spoil the stew. One module can change the cur-

Listing 5 An optimizer might make incorrect assumptions regarding *x/y* and *i*4*.

```
float array[256];
calcarray(x,y)
float x,y;
{
 unsigned char i;
 /* put ratio in every 4th cell in array */
 for (i=0; i<64; i++)
 {
  if (y != 0}
  array[i*4] = (x/y);
 }
}
```

rency of a global variable and cause another module to do something unexpected later.

These types of errors require the programmer to rethink the layering of the program at a modular level. A source-level debugger may be needed to understand how the interactions occur and to hack in a fix, but the real solution might well lie in the program architecture.

Interrupts can present their own set of difficult bugs. An obvious example of an interrupt bug occurs when the interrupt neglects to save or restore the entire status of the machine before returning, as often happens when modifications have been made to an interrupt routine. If the modification uses a register that wasn't saved during the interrupt prologue, then any routine in the foreground program that also uses that register is vulnerable to the interrupt. This oversight can cause previously working code to break.

An interrupt routine may call a function that isn't reentrant (a problem if the foreground program also uses that function). Library routines, particularly in floating-point math libraries, may be reentrant for one brand of compiler but not for another.

Generally, interrupt routines require that a debugger be able to deal with code at the assembly language level. In addition, the debugger must be able to operate transparently to the interrupt and vice versa. In multithreaded or multitasking systems, integration bugs can breed and multiply. Shared resources must either have lock semaphores or be designed to be reentrant.

Processes that `malloc` memory or open files must free the resource when they're done with it; otherwise, the system will eventually—maybe even two or three days later—run out of memory or file handles and lock up. Programs employing `setjump/longjump` and `goto` statements must be carefully designed to avoid abnormal control flow that may leave these resources tied up.

A cousin of the typematch bug is the portability bug. This bug surfaces during porting from one machine to another and can cause both intent and execution errors. Since C is implemented in different ways on different machines, tricks that work on one machine may not work on another.

The quickest way to debug a program is to write a program that has no bugs.

Variations in the size and alignment of various objects, particularly differences in the implementation of types `float` and `double`, can also cause problems. Segmented microprocessors may treat pointers differently from nonsegmented machines; stackaddressing direction and byte ordering can also be different. Programmers with limited experience writing portable C code will undoubtedly discover these pitfalls the first time they compile their old programs on a new machine.

Compiler bugs are rare, but they do occur. All too often, the bug isn't really the fault of the compiler but of poor coding practice. Many of the newer compilers on the market perform code optimization, and the optimizer may make assumptions about the program that aren't desired. This problem is further complicated by the fact that debugging is usually performed on unoptimized objects, while the debugged program is compiled using the optimize mode. If bugs show up after optimizing, then the optimizer is probably the culprit.

Optimized code, especially globally optimized code, is tougher to debug since the object program may not match the source code line for line. Also, some source-level debuggers won't even allow optimized code to be debugged, forcing us to retreat to our assemblylevel debugger for support.

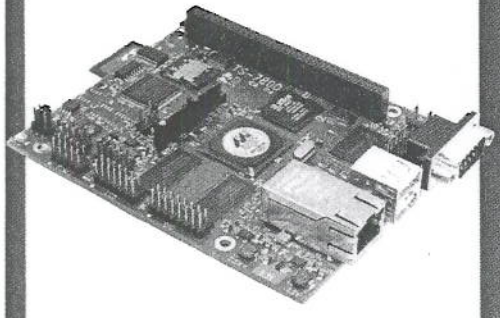
The most common optimizer bug is caused by memory-mapped I/O. Consider an I/O device memory-mapped at address `data_port`. The function in Listing 4 waits for ready status, writes a sequence of characters to the device, then returns the new ready status to the caller.

An optimizer could just have a ball with this. First, it might consider that the `while` loop was a redundant read of the same address and replace it with a single read of the location. Next, it might determine that there was nothing to do in the body of the `while` anyway and decide that the entire `while` statement wasn't even needed. It would probably assume that the A and C assigned to the address were dead stores and replace them with the single assignment `*data_port = 'K'`. As for the return, it might assume that the last thing written to the address, a K, should still be there and simply return K instead of reading

Listing 6 A possible result of optimization of the Listing 5 code: an endless loop and a divide error.

```
float array[256];
calcarray(x,y)
float x,y;
{
 unsigned char i;
 register float tmp;
 tmp = x/y;
 /* put ratio in every 4th cell in array */
 for (i=0; i<256; i+=4)
 {
  if (y != 0)
   array[i] = tmp;
 }
}
```

TS-7800 High-End Performance with Embedded Ruggedness



$269 qty 1 $229 qty 100

500 MHz ARM9

- New unbrickable design- 3x faster
- Backward compatible w/ TS-72xx
- Low power - 4W at 5V
- 128MB DDR RAM
- 512MB high-speed onboard Flash
- 12K LUT user-programmable FPGA
- Internal PCI Bus, PC/104 connector
- 2 USB 2.0 480 Mbps
- Gigabit ethernet
- 2 SD sockets
- 10 serial ports
- 110 GPIO
- 5 10-bit ADC
- 2 SATA ports
- Sleep mode uses 200 microamps
- Boots Linux in < 2 seconds
- Linux 2.6 and Debian by default

Design your solution with one of our engineers

- Over 20 years in business
- Never discontinued a product
- Engineers on Tech Support
- Open Source Vision
- Custom configurations and designs w/ excellent pricing and turn-around time
- Most products stocked and available for next day shipping

See our website for options, peripherals and x86 SBCs

Technologic SYSTEMS

We use our stuff.

visit our TS-7200 powered website at

www.embeddedARM.com

(480) 837-5200

the status. Now the code, if represented as C source, looks like this:

```
int sendack()
{
extern char *data_port;
*data_port = 'K';
return((int)('K'));
}
```

where `*data_port` is the address of the device.

Because optimizers may rearrange the code to minimize computation, the only safe way to avoid optimizer errors when dealing with memory-mapped I/O is to compile the driver with optimization turned off. This practice is usually safe but in special situations will cause bugs. When optimizing the function in Listing 5, for example, an optimizer might make two assumptions: that the ratio *x*/*y* could be done one time, and hence calculated before the loop, and that the multiply operation in the array index computation `i*4` could be avoided if the loop were written differently. The optimizer may produce something equivalent to Listing 6.

The function now has two bugs! First, the loop will never terminate because `i` is an unsigned character and can't reach 256; second, a divide error will occur if y==0. Some compiler optimizers are smart enough to detect these situations and avoid producing code with these bugs.

SUGGESTIONS FOR WRITING BETTER C

The quickest way to debug a program is to write a program that has no bugs. Software that's modular and nicely layered will usually have fewer integration bugs. Here are some suggestions for producing code with a minimum of problems.

- Be extremely careful when using pointers; uninitialized-pointer bugs and boundary bugs can be very time consuming to correct.
- Look for typematch bugs by using `lint` or other utilities, or simply do a "paper debug" to check each function call for proper argument and return types.
- Be careful when using macros, especially those containing parameter substitutions. Capitalizing all macro names can serve as a reminder that they're macros, not function calls.
- Use parentheses liberally to guarantee associativity.
- Think about portability while writing code. If necessary, include a header file containing `typedefs` for basic types (`BYTE`, `WORD`, etc.), then use these instead of `char` and `int`. Programs can then be ported to another machine or compiler by modifying the `typedefs` in the header file.
- Use casting when converting types; don't expect the compiler to do it for you.
- Avoid global variables. (within reason)
- Use header files for function prototyping and argument definition.
- Use return codes for modules that interface with each other.
- Above all, design the tests to exercise all branches in the program. Include tests for boundary conditions, especially for code using pointers. If possible, keep a test suite for future use, should the module ever be modified or ported to another environment.

Writing software is a complicated and tedious puzzle. Even with a concerted effort by the programmer, it's nearly impossible to produce a nontrivial program that's bug-free the first time. Knowing the potential causes of bugs allows us to adopt disciplines to minimize their occurrence and guides our efforts to stabilize, localize, and correct them. ■

Robin Knoke cofounded Applied Micro-systems Corp. When this article was written, he was involved in the specification and design of productivity tools for creating and debugging embedded systems software. He left AMC in 1990 and is now running the White Salmon Group, Inc. a small product-development company. ("Small and I like it that way," he says.). White Salmon Group (*www.whitesalmongroup.com*) designs embedded microprocessors for several clients—and Knoke still does a lot of C coding. You can reach him at knoke@whitesalmon.com.