Rules for defensive C programming

by Dinu Madau
Software Engineer
Visteon Automotive Systems

Software professionals have learned over the years that software is prone to illness. I use the term "illness" instead of "software error" or "failure" because software inherently doesn't fail. Given the same operating conditions, the software will behave in a predictable manner. The behaviour may not be desirable and may induce a failure in the overall system but the software itself does not fail. Contrast that with a failure in a mechanical system due to fatigue. The software may have been coded incorrectly or coded correctly to an incorrect requirements specification. The symptom of the illness can be partial or complete system failure and is often hidden for most operating conditions. After the software has undergone verification and validation, it is considered healthy and ready to be released to the public. All too often it is only when the product is in the hands of the customer that the symptom of the illness becomes evident.

Neither is the compiler immune to symptomatic illness, since it is a complex software system. Initially, C compilers on the market had characteristics that were often unique to themselves. After all, compiler writers had no common standard that fully defined their product's operation. Even when a programming standard was defined by the C standards committee, some compiler writers, particularly those for embedded systems, continued to treat the standard as the committee's opinion of how their product should behave. What amplifies this problem is that the committee defining the standard could not even agree on some of the behaviours of the C programming language (for example, Annex G ISO/IEC 9899-1990 unspecified behaviour). 1

Table 1 Reverse notation table

Standard C Types		
Prefix	**Type**	**Example**
uc	unsigned char	ucByteVariable
sc	signed char	scByteVariable
bf	bit field	bfFlags
us	signed short	ssWordVariable
ss	signed char	scByteVariable
ul	unsigned long	ulDouble
sl	signed long	slDouble
adt	abstract data type	adtUNION_VAR
fl	floating	flSpeedOfLight, flSHUTTLE_TRAJECTORY
Pointers to any of these types (add a p in front of the prefix)		
Prefix	**Type**	**Example**
p???, p??	pointer to ??? type	pucByteVariable (a pointer to an unsigned character type) padtSTRUCTURE (a pointer to an abstract data type)
pa??	pointer to array of type ??	paucByteVariable (a pointer to array of type character)
Scope		
Prefix	**Type**	**Example**
All uppercase	global variables	ssGLOBAL_VAR_ONE
First letter uppercase and at least one lowercase letter to follow	local variables	usLocalOne, slLOCALone, bfFlags_4TEST
Constants		
Prefix	**Type**	**Example**
All uppercase with optional prefix type followed by an underscore	constants	TIME_CODE, SS_PI, optional prefix type SL_GRAVITY

The intent of this article is to increase the quality of the software product by defining a verifiable and enforceable defensive programming standard for your software development team. This standard should take into consideration the limitations of the C coding standard and your compiler. Develop a habit of using defensive programming techniques.

What's a software professional to do in this complex world of compilers, committees, and standards? If we consider the analogy of driving a car, we can parallel the methods used to avoid an automobile accident with software engineering techniques. Automotive manufacturers develop vehicles that are inherently stable and predictable. On the other hand, the environment in which these vehicles are used is less predictable, and the users of the vehicles are prone to error due to a number of preoccupations—after all, we're only human. To reduce the possibility of an accident, local governments have established laws that must be followed. We obey the laws and as an additional precaution, develop a defensive driving style to avoid accidents. If we are involved in an accident, the casualties are typically less than those resulting from an offensive driving style, otherwise known as road rage.

Let's transfer this analogy to the world of software development. The various compiler behaviours are usually well defined, even for behaviours that are undefined in the standards. The user/programmer's environment is less predictable due to a number of real-world preoccupations such as deadlines, schedules, stress, or co-workers in the next cubicle. The software standards committee realised that a "law" had to be established. After

much deliberation, the American National Standards Institute (ANSI) C standard was established, which today is referred to as the ISO 9899-1990 standard for programming languages—C. People, unfortunately, are still prone to error, which leads to the necessity of a defensive programming style to reduce the possibility of an "accident" or even worse, the nasty f-word: "failure." Arrogant programmers assume that compilers have the ability to read their minds and determine the intended functions of their code fragments. Allow me to clarify with an example.

Note the misuse of operator precedence in the following statement, a common file handling routine:

```
if (InVar = getc(input)
!= EOF)
```

The programmer's desire is that the next character in the input stream from the file be loaded into the variable InVar and compared with the end-of-file character (EOF). The only problem is that the order of precedence of the compiler was assumed by the programmer (unfortunately, the compiler can't read the mind of the programmer) and the function executes as follows:

```
if (InVar = (getc(input)
!= EOF))
```

InVar will either be set to TRUE or FALSE but never to the character of the getc command as is desired by the "offensive" programmer. In fact, InVar is set to TRUE only when the EOF marker is reached at the end of the file. This statement of code is then reduced to an end of file "finder." (Another inventive use for this line of code is to induce a variable delay into your system, dependent on the length of your input file.)

The goal of this article is to introduce you to methods that can help reduce the occurrence of software illness. I will discuss different techniques for defensive C programming that have been used in development of safety-critical systems. Each section will establish a guiding rule or principle followed by supporting material, which in most cases will include examples, pseudocode, and C code.

Standardisation

Standardisation leads us to the first rule of defensive C-ware: Every software development team should have an agreed-upon and formally documented coding standard. A coding standard consists of a set of rules that addresses weaknesses in the language standard and/or compiler idiosyncrasies and also defines a format or "style" used for writing code. Typical items in a coding standard could address pointer usage before initialisation, the use of recursive algorithms, dynamic memory allocation, unconditional jumps, and so on. The coding standard should also help to improve readability. Every person developing software has unique coding practices. Therefore, if more than one person is working on a project, a common standard should be established. It's like handwriting—you may be able to read your own chicken scratch but others most likely will struggle through it.

One of the reasons for developing a coding standard is to make code more readable, which will positively affect the following areas in software development:

- Code generation. A standard reduces the probability of coding error
- Code reviews. A standard increases the efficiency of peer reviews and inspections
- Quality. A standard increases the overall quality of the product
- Maintenance. A standard increases the maintainability of the software product

If you've ever attended a meeting for developing a standard, you'll notice that people are quite attached to their own personal style of programming, and that trying to get them to buy into your style is like telling them that their babies are ugly. When you attend these meetings, be prepared to discuss items that will affect the goals for developing the standard. If you have a personal style that you want to provide which does not add to the underlying goal of making software more readable, then it's best to leave that baby at home.

Table 2 Precedence table for C

Operators	Associatively
( ) [ ] -> . ++ --(post)	Left to right
! ~ (pre)++ -- + - * & (type) sizeof	Right to left *
/ %	Left to right
+ -	Left to right
< < >>	Left to right
< < = > >=	Left to right
= = !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
?:	Right to left *
= += -= *= /= %= &= ^= |= < < = >>=	Right to left *
,	Left to right

*Reversed associate property

If you don't have a good standard in place, start off by using one of the publicly available standards. At first, try to use portions of the standard that will give you the highest return on investment and slowly expand the coverage as applicable for your development. Change is best implemented in small increments; otherwise you'll probably encounter large resistance to it. Standards already exist for specific market segments, such as the Motor Industry Software Reliability Association (MISRA, April 1998) intended for embedded automotive applications. The standard consists of 127 rules, 34 of which are defined as advisory. It is available on the Internet at www.misra.org.uk . Another standard developed by the U.S. Nuclear Regulatory Commission (June 1996) is the NUREG CR-6463, entitled Review Guidelines on Software Languages for use in Nuclear Power Plant Safety Systems . Both of these standards are good starting points.

A standard is only as good as the ability of the team to verify and enforce its implementation. It is a futile task to take good software professionals and get them involved in software politics by creating laws or standards that are not only unenforceable but also unverifiable. Therefore, someone with authority (that is, the person who writes the performance reviews) must support your efforts, or they will fail.

A preferred method for verifying compliance is through the use of static checking tools. These tools, although powerful, are usually unable to guarantee full compliance (depending on the details of your standard). Another way to verify and enforce compliance is through code reviews and inspections. 2

Variable naming . One of the items that is usually discussed at

a coding standards meeting is a naming convention for the various C-types. The goal is to be able to look at a line of code and determine the variable or constant types used. Take the following line of code, for example:

```
Energy = Mass *
(Speed_Of_Light_
squared);
```

You have limited information about this equation without referencing the definitions document for the variables and constants used in the formula. The task of trying to inspect this line of code with respect to balancing and promotion of types would be time consuming and prone to error due to cross referencing. Determining if variables are local or global, or whether any concern exists about overflows in the calculation, can be difficult.

Now assume the following abbreviated variable naming convention:

- All variables will have a lowercase, two- to three-character prefix which defines its type followed by an uppercase character for the start of the variable name (ss = signed short, sl = signed long, uc = unsigned character, and so on)
- All local variables will have intermixed uppercase and lowercase characters to define their names
- All global variables will be defined in uppercase
- All constants will be uppercase and if typecast in the #define, then a two-character prefix type in uppercase followed by an underscore will be used to define their type

Following these rules, we would rewrite the previous equation as:

```
slENERGY = usMass *
(SL_SPEED_OF_LIGHT_
SQUARED);
```

An analysis of the this line of code reveals that slENERGY is of type signed long and is a global variable. SL_SPEED_OF_LIGHT is a constant cast in the define as a signed long. The variable usMass is a local variable of type unsigned short. A check of balancing and promotion is easily done now that the variable types are readily identified. The product of usMass with the SL_SPEED_OF_LIGHT forces usMass to be cast to an intermediate type signed long integer. The advantages of a naming convention are clear. The information now available from the rewritten line of code far outweighs the information available from the original line of code.

Now when you see the following statement at a code review:

```
ssPRODUCT = (ssNumber1
*
ssNumber2) / scDivisor;
```

you should question whether the intermediate product would fit in a signed short or just play it safe and cast the intermediate product to a signed long before dividing. Casting one of the elements of the product to a signed long forces the intermediate product of ssNumber1 and ssNumber2 to be promoted to a signed long:

```
ssPRODUCT = ((signed
long)ssNumber1 * ssNum-
ber2) /
scDivisor;
```

Let's define a naming convention for C types. The reverse notation uses a prefix to define the variable's type. Table 1 contains conventions for all of the C types, in reverse notation.

Table 3 Limiting the definition of macros

Macro name	Definition	Description
#define SIGN(x)	(((x)>=0) ? (1) : (-1))	Sign of variable i
#define MAX(x, y)	(((x) < (y)) ? (y) : (x))	Max of x or y
#define MIN(x, y)	(((x) > (y)) ? (y) : (x))	Min of x or y
#define LIMIT(x,low,high)	((x) >(high)?(high):((x)<(low)?(low):(x)))	

Table 4 Defining replacements for standard C types

Replacement type	C-Types	Compiler sizing
typedef unsigned char	UBYTE;	/* 8 bit types */
typedef signed char	BYTE;	
typedef unsigned short	UWORD;	/* 16 bit types */
typedef signed short	WORD;	
typedef unsigned long	ULONG;	/* 32 bit types */
typedef signed long	LONG;	

Magic numbers

Do not use hard coded numbers directly in expressions. Maintainability becomes a horrendous task when the code is contaminated with magic numbers. A magic number has meaning to the original programmer at the time of coding, but when the programmer leaves the company due to an evolutionary process called career development, the significance of the number (unless defined in a comment or definition table) can lose meaning. Let's take the simple line equation to help illustrate this point. The equation is:

```
y
 =
m
 *
x
 +
b
```

where m is the slope of the line, b is the y intercept (at x = 0), and x (input) and y (output) are the points on the x - and y -axes. You can write the following line of code to satisfy the requirements:

```
ssY = (31.872983 * ssX)
+ 45.3232;
```

If this line of code is out in the field for many years and then a requirement change is issued forcing a modification to the code, at first glance a person would try to determine the origins of the numbers. If the code is properly commented, this becomes less of an issue (although the same constant might be used in a different part of the software, increasing complexity). Contrast that with the following line of code:

```
ssY = (SS_PRESSURE_
SLOPE * ssX) + SS_PRES-
SURE_INTERCEPT;
```

The first observation you can make is that the equation deals with some type of pressure constants. The key words SLOPE and INTERCEPT indicate to the reader that the expression might take on the format of a line equation. The constants can be defined in a parameter or constant definition file or in the local header file. Any changes in the value of the constant would propagate throughout the definition scope of the software product, sparing the programmer the burden of searching the entire software product for all occurrences of the constant.

Indentation

Establish an indentation standard and be consistent throughout the work product. Structured indentation makes software easier to read, thereby reducing the possibility of coding error. Consider the following example:

```
for (x=0; x
<
= LIMIT; ++x) {
for (y=1; y
<
```

```
= Y_LIMIT; ++y)
{
z+= x*y;
if (x == TEST) {
printf("x value pass");
t=x/AVERAGE;
}
z /= 4;
}
}
```

Nerves of steel would be required to prepare for a code inspection if you had to review page after page of poorly indented code. This piece of code could give the visual illusion that the line of code z /= 4; executes as part of the x for loop. Remember, you're a software professional, not a magician. Leave the illusions to management.

Now consider the more palatable solution:

```
for (x=0; x
<
= X_LIMIT; ++x)
{
for (y=1; y
<
= Y_LIMIT; ++y)
{
z += x*y;
if (x = =TEST)
{
printf("x value pass");
t = x/AVERAGE;
}
z /= 4;
}
}
```

Good indentation helps to visually define the structure of the code. The line of code z /= 4; is clearly executed as part of the y for loop. In this example, all open and closed braces were put on separate lines with the open braces lining up with the first character of the conditional/loop expression. Both versions of the source code execute identically; the only difference is the visual interpretation. A good rule of thumb is to allow only one statement per line of code and insert braces on their own line.

For complex conditional statements with multiple elements, put each element on its own line. See Listing 1 for an example.

Although the conditional statement in Listing 1 can be formatted in other ways that would equally increase the readability of the expression, placing each condition on separate lines has additional advantages (similar to those of placing each variable definition on its own line). A change to one condition of the multi-conditional statement will not affect the other conditions. Each line can then be individually commented for enhanced clarity, as shown in Listing 2 .

You be the judge. Which piece of code would you rather inspect? Whatever form of indentation you decide to use, you must be consistent throughout your code. Remember, one of the objectives is to make the code more readable.

Parenthesizing

Always use full parenthesizing for macros and equations. To reduce the possibility of error when using the preprocessor, always use full parenthesizing. Most people who use parentheses when defining negative numbers don't use them when defining positive numbers. Parenthesizing all numbers is beneficial for the following reasons. First, nobody has ever gotten into trouble by using too many parentheses; it's the placement of the parentheses that introduces errors. Second, let's consider the following example of defining a positive number:

```
#define POS_NO 10
```

When the preprocessor executes it will find the location of POS_NO and replace it with the value of 10. Since the operation of the preprocessor is weakly defined, the compilation of the following line of code is entirely dependent on the compiler:

```
ssY = .POS_NO;
```

The decimal point before POS_NO might be treated as just that, a decimal point and the line would appear to the compiler as:

```
ssY = .10;
```

If you're fortunate, your compiler would freak out at this implementation and generate an error message (in capital letters because it likes to shout when you do stupid things). This is the best case scenario. The compiler might very well compile this line of code and grant you the luxury of finding the problem in the debugging phase. The whole problem could be easily avoided by #defineing the POS_NO parameter in parentheses:

```
#define POS_NO (10)
```

In all cases, the preprocessor would replace POS_NO with (10) and the code would appear to the compiler as:

```
ssY = .(10);
```

In this case the compiler should flag an error. It's worth noting here that the period or decimal point (.) is defined in the C language as an operator with similar attributes and constraints as other operators.

Precedence . If you've memorised the precedence table for C and trust your memory, you can skip to the next section. The rest of us will take a look at the order of arithmetic and logic precedence for C. Table 2 is arranged from high to low precedence. Operators on the same row are of equal precedence and their associative property is defined in the right column.

For three levels of precedence, the associative properties are reversed from the rest of the table. The "equal" (=) assignment operator is treated as any other operator in C, unlike other programming languages (such as Pascal and Ada) in which the assignment operator is unique.

Consider the following equation:

```
ssY = ssM * ssX + ssB;
```

Assuming that the algorithm author's intent is that the multiplication should execute first, the line of code should be written as:

```
ssY = ((ssM * ssX) +
ssB);
```

By full parenthesizing every expression, you'll eliminate the question of when to and when not to do it. It also aids in documenting the programmer's intent. Maybe the programmer wanted the sum of ssX and ssB to be executed before the multiplication. The equation would then be written as:

Listing 1 Complex conditional statements with multiple elements:

a) Every element on the same line;
b) Each element on its own line

```
a)
if (((ssFL_Valve==FRONT && ssCOIL_BIAS==FRONT) || (ss-
CTRL_Valve==REAR &&
ssCOIL_SIDE==LEFT || ssCOIL_SIDE==RIGHT)) && ssPOWER_
FROM_IGNITION!=OFF)
b)
if
(
(
(
ssFL_Valve==FRONT
&& ssCOIL_BIAS==FRONT
)
||
(
ssFR_Valve==REAR
&& ssCOIL_SIDE==LEFT
|| ssCOIL_SIDE==RIGHT
)
)
&&
ssPOWER_FROM_IGNITION!=OFF
)
```

```
ssY = (ssM * (ssX +
ssB));
```

Full parenthesizing should be second nature, especially in a safety-critical system. Too many software bugs are a direct result of incorrect precedence assumption and/or poor parenthesizing. The solution for eliminating precedence problems is easy: use full parenthesizing.

Using the preprocessor

Don't use the preprocessor for defining complex macros. Of course, the word "complex" is open for interpretation. It should therefore be defined within your development team. The operation of the preprocessor is poorly defined in the C coding standard, so its operation is at the mercy of the compiler writers. The preprocessor does have some valid uses, which, when applied properly, can help increase the maintainability and readability of the code.

The preprocessor is useful for defining "manifest constants" and for pre-calculating constants. This feature allows the user to force the preprocessor to calculate mathematical expressions into a constant type. The value of that type then gets replaced throughout the scope of the code. Scaling and conversions are common uses for this feature. For example, the conversion from miles per hour (MPH) to kilometers per hour (KPH) is required for a vehicle parameter that is given in miles per hour:

```
#define MPH_TOP_VEHICLE_
SPEED  (154)
/*MPH*/
#define KPH_TOP_VEHI-
CLE_SPEED
(MPH_TOP_VEHICLE_SPEED
* 1.6)
/*KPH*/
```

The calculation is carried out using the highest precision of the preprocessor, usually floating point, and can then be cast to the required type. Despite the simplistic example, one can envision multiple conversions and scalings within one #define . When the parameter for top vehicle speed changes, the intermediate constant calculations are automated by the preprocessor and the change would be propagated throughout the scope of the code.

Let's consider another example. We will allow the preprocessor to do calculations in floating point and then convert the constant to the format in which it will be used. The formula for the area of a triangle is half of the base times the height. Let's assume that for this application, the base and height are known system parameters. The base is equal to 10cm. The height is equal to 5cm. You could either code the area of the triangle directly:

```
#define TRIANGLE_AREA
(25)
```

Or you could define the base and height separately:

```
#define
TRIANGLE_BASE (10.0)
/* cm */
#define TRIANGLE_HEIGHT
(5.0)
/* cm */

and then define TRIAN-
GLE_AREA as:

#define TRIANGLE_AREA
(0.5 *
TRIANGLE_BASE * TRIAN-
GLE_HEIGHT)
```

The value, TRIANGLE_AREA , is calculated by the preprocessor with the maximum precision available. TRIANGLE_AREA can then be type cast in the code or in the #define as follows:

```
#define US_TRIANGLE_AREA
(unsigned
short)  (0.5 * TRIANGLE_
BASE *
TRIANGLE_HEIGHT)
```

Since the area will always be positive, an unsigned short type is used. The advantage to this method is that if the base or height of the triangle changes, you don't have to recalculate the area. All you have to do is change the parameters of the triangle in the definition. This technique becomes highly valuable when you have constants that are a direct result of complex formulas based on physical system parameters.

Listing 2 Each line can be individually commented for clarity

```
if ((( ssFL_Valve==FRONT /*FL valve set to front when
coil is set to front*/
&& ssCOIL_BIAS==FRONT /* Coil Bias: see DFD 1.3.4 for
more information */
)
||
( ssCTRL_Valve==REAR
/* Control valve set to rear check coil side */
&& ssCOIL_SIDE==LEFT
/*Assure that one side is selected. Left
Side check*/
|| ssCOIL_SIDE==RIGHT
/* Check if Right Side is selected */
)
)
&& ssPOWER_FROM_IGNITION!=OFF /*Make sure that we are
not in power save mode*/
)
)
```

Limit the definition of macros in the preprocessor to highly reuseable, simple functions. See Table 3 for an example.

Notice the use of full parenthesizing, which ensures that the arguments and the resulting expressions are properly bound. There is no room for ambiguity. The previous macro definitions are generic functions that will most likely be used often. Defining macros alleviates the need for a function call, which reduces execution time. However, taking the address of the macro is undefined because it has no address. Unique, application-specific functions should not be defined as macros.

#IFDEF . A word of caution for those who use #ifdef instead of #if . A common use for the #if statement is to strap in and out portions of code that are configuration dependent. It would seem tempting to use the #ifdef , which calculates to TRUE if the variable is defined. Consider the following example.

In a configuration header file, the following define statements exist:

```
#define FOUR_WHEEL_DRIVE
(0)
#define REAR_WHEEL_DRIVE
(0)
#define FRONT_WHEEL_
DRIVE  (1)
```

The configuration file allows the programmer to choose between the three different vehicle configurations that are available for the software product. Each configuration consists of unique straps or patches that must be included at compile time. Now suppose the programmer wants to strap out a piece of code using the defined configuration as follows:

```
#ifdef REAR_WHEEL_DRIVE
{do code block A}
#endif
#ifdef FRONT_WHEEL_
DRIVE
{do code block B}
#endif
```

Both blocks of code would be included at compile time because the #ifdef only interrogates the variable name for a definition. A definition of zero (0) would still flag a logic TRUE to the preprocessor. Avoid this by eliminating the use of the #ifdef and #ifndef and use #if instead. A #undef operator is provided to undefine a variable if for some reason you crave the use of the #ifdef operator.

Macros for the interface . Another good use for macros is in developing reusable interfaces. For example:

```
#define ssGET_SENSOR_
SIGNAL()
(signed short
int)(RAW_ANALOG_INPUT.
SENSOR1)
#define ssGET_NEVRAM_
DATA ()
```

```
(NVRAM_STRUCTURE.DATA_
REGISTER)
#define
ssPUT_NVRAM_DATA ( x )
(NVRAM_STRUCTURE.DATA_
PORT = (x) )
```

The interface is defined by Get and Put macros and the definition of these macros are hardware-, machine-, and/or architecture-dependent. In this example, the ssGet_sensor_signal macro looks for a data structure (RAW_ANALOG_INPUT) that is memory mapped to the A/D converters and picks out the SENSOR1 port of the A/D converter. As an algorithm developer, my only concern is to get the sensor data from the first sensor (SENSOR1). This means that what is hidden or encapsulated into the macro (similar to C++) is of little interest once the interface is designed. The sensor data might come from a telemetric signal that is bounced off of a satellite and then uploaded through the World Wide Web via an HTML page. It does not really matter. What's important is the value of the sensor signal at a given time. Therefore, the interface in the main code would consist of Get and Put macro statements. When the hardware architecture changes, only the macro definition is modified to correspond to the change. The software is completely void of any interface code with the exception of an interface header that links the outside world to the inside.

Macros are useful when applied correctly. Conversely, they can be your worst enemy, unmercifully enslaving you to agonizing hours of debugging. Ignorance is not bliss in this case.

The balancing act

Most of the time we don't care how the compiler internally balances and promotes the different variable types, as long as we get the desired result. When we don't get the desired result we are rudely awakened to the tune of assembly-level debugging, trying to determine why a simple, straightforward equation does not function correctly. In C, the promotion rule is as follows:

Integral types of character, short integer, or an integer bit field and their signed and unsigned varieties will be converted to signed integer if the signed integer can represent all values of the original type; otherwise it is converted to unsigned integer. (6.2.1.1 ISO)

So chars, shorts, and bit fields get converted to some type of integer (signed or unsigned) before an expression is executed internally.

To avoid potential problems, explicitly cast non integer operands in expressions. When using char, short, or bit fields types in expressions, always cast them to the appropriate integer type to reduce the possibility of conversion errors. If you don't, C automatically converts these integral types to integer before executing the expressions, and the conversion might not give the desired result. Even if you verify the behaviour of your compiler for conversions of these types, when you switch to another compiler for a different program, its behaviour might be different. Let's look at another example that will help illustrate the problem.

Let UC be a variable of type unsigned char set to the value of zero (0). Will the following conditional expression result in true or false?

```
UC = 0;
if (UC == ~0xFF)
```

From the C balancing rule, both UC and the hex value 0xFF are converted to type signed integer (because their values can be contained in the signed data type) before being evaluated. UC is cast to signed short 0:0 (both bytes set to zero). The hex value 0xFF is cast to a signed short 0:FF (the first byte is 0 and the second byte is set to hex FF). The not operator functions on the integer type, which results in FF:0. The comparison is then done between 0:0 and FF:0 and the expression from the view of the compiler takes on the following after the negate operator is done:

```
if ((signed short)0x0000
= =
(signed short)0xFF00)
```

This expression would result in an evaluation of false. The problem could easily be avoided by casting the constant to the desired type:

```
if (UC == (unsigned
char)~0xFF)
```

This will effectively truncate the top byte of the integer and the comparison will be done on character types. When in doubt, cast it out.

Explicitly define your types as signed or unsigned. If you don't specify the signed or unsigned type, the C compiler will select one for you, which might not achieve the desired result. In most cases, when defining a variable as an integer, the compiler assigns the default type of signed short integer unless the value cannot be contained by that data type. When defining a bit field or character type, the default type is entirely compiler dependent. Explicitly defining your data types as signed or unsigned increases the portability of the code. What you might consider to be an unsigned character type in your code might actually be viewed by the compiler as a signed character type. By allowing the compiler to choose for you, you're essentially giving up your first amendment right as a software developer, which states that software professionals are more capable than compilers at making decisions. What about a signed bit field of length one?

Explicitly cast mixed precision arithmetic operands in expressions. In expressions, the subexpressions are evaluated at the appropriate operand precision. The desired result may not be achieved if the resultant precision is greater than the expressions operand precision. To eliminate this error, explicitly cast the operands to the final precision of the result. For example:

```
signed short ssV1 = 1;
signed short ssV2 = 2;
float fResult;
fResult = ssV1 / ssV2;
```

Here fResult incorrectly calculates to zero because the sub-expression is evaluated with integer precision. The expression:

```
fResult = (float) (ssV1 /
ssV2);
```

also incorrectly evaluates to zero for the same reason. The subexpression ssV1 / ssV2 is calculated with integer precision and then the integer result is cast to floating precision. Consider the following:

```
fResult = (float) ssV1 /
ssV2;
```

or:

```
fResult = (float) ssV1 /
(float) ssV2;
```

Both evaluate to the correct result (0.5). Explicitly casting one of the operands in the sub-expression to the desired precision of the result forces the sub-expression to be evaluated at the resultant precision.

Portability

Reusable code—everyone talks about it, yet when you're asked to reuse someone else's code, a chill runs up your spine. Let's face it, at one time or another most of

Listing 3 Macros for rounding, with a type cast thrown in

```
#define UC_RND(i) ((UBYTE)((i)+0.5))
#define SC_RND(i) (((i)>=0) ? (BYTE)((i)+0.5) :
(BYTE)((i)- 0.5))
#define US_RND(i) ((UWORD)((i)+0.5))
#define SS_RND(i) (((i)>=0) ? (WORD)((i)+0.5) :
(WORD)((i)-0.5))
#define UL_RND(i) ((ULONG)((i)+0.5))
#define SL_RND(i) (((i)>=0) ? (LONG)((i)+0.5) :
(LONG)((i)-0.5))
```

us had to reuse a piece of code that was supposed to be portable. In reality, it wasn't. And we all remember the joy of debugging someone else's code. This section will address some guidelines for designing code that is targeted for portability. One of the weaknesses of the C standard is in defining the sizes of types. C specifies that:

```
char >= 8 bits
short >= 16 bits
integer >= 16 bits
long >= 32 bits
and a char <= short <=
integer <= long
```

Portability of types becomes an issue in the C language because the compiler writers must determine the size of types based on these constraints. The solution is to define replacements for standard C types. Table 4 can be defined in a header file and then modified to reflect the size differences of your C compiler.

The replacement types are used throughout the code for casting and defining variables. When you switch to a different compiler, just change the header file to reflect the type sizes.

Defining logical opposites . How many times have you seen the following?

```
#define TRUE    (1)
#define FALSE   (0)
```

When defining logical opposites, first define one of the logical states and then define the opposite state as a macro based on the original state:

```
#define TRUE    (1)
<
#define FALSE   (!TRUE)
```

This method has two advantages. First, if the logical state for TRUE is changed, the logical states that are based on TRUE are automatically changed. By design, this increases the readability of the code, provides a better understanding of the intent of the original software writer, and helps document the code. The software engineer doesn't have to search and guess which states are logical opposites. Second, some optimisers will perform better if the logical opposite states are based on each other. An example that isn't totally intuitive would be defining a flag that indicates bit ordering:

```
#define MSB_First (TRUE)
/* most
significant bit is
packed first */
#define LSB_First (!MSB_
First)
/* least significant bit
is
packed first */
```

In this case LSB_First is the logical opposite of MSB_First because only one can be true for any given configuration. Therefore, their definitions are based on each other.

Bit ordering . Bit-field ordering is compiler dependent. There are two common bit field orderings: most significant bit (MSB) first or least significant bit (LSB) first. When defining a bit field variable, use the preprocessor directive #if to strap in the desired bit field ordering at compile time based on a configuration flag. Some compilers will give you a command line option to reverse bit ordering. Relying on the compiler option, though, reduces the portability of the code. For example, let's look at the following definition:

```
#if
(MSB_First)
struct Flags {
unsigned int  Bit7: 1;
unsigned int  Bit6: 1;
unsigned int  Bit5: 1;
unsigned int  Bit4: 1;
unsigned int  Bit3: 1;
unsigned int  Bit2: 1;
unsigned int  Bit1: 1;
unsigned int  Bit0: 1;
};
#elif (LSB_First)
struct Flags {
unsigned int  Bit0: 1;
unsigned int  Bit1: 1;
unsigned int  Bit2: 1;
unsigned int  Bit3: 1;
unsigned int  Bit4: 1;
unsigned int  Bit5: 1;
unsigned int  Bit6: 1;
unsigned int  Bit7: 1;
};
#endif
```

If MSB_First gets set to TRUE (1) , the bit ordering of the first structure is used. If LSB_First is set to TRUE , the bit ordering of the second structure is used.

Max and min limits . If you don't have a limits.h file for your compiler, define the limits of your types in a separate header file. Even if you have a limits.h file, it would be good to redefine your limits based on your defined types. If your limits change due to type size changes, you need only change the header file for these limits. A typical limits file definition based on the replacement types of Table 4 consists of the following:

```
#define BYTE_MIN  (-127)
/* 8-bit
type */
#define BYTE_MAX  (127)
#define UBYTE_MAX  (255)
#define WORD_MIN  (-32767)
/* 16-
bit type */
#define WORD_MAX  (32767)
#define UWORD_MAX  (65535)
#define LONG_MIN  (-
2147483647)
/* 32-bit type */
#define LONG_MAX
(2147483647)
#define ULONG_MAX
(4294967295)
```

Floating point to integer

When you allow the macros to pre-calculate constants using the maximum resolution of the preprocessor, which is typically done in floating point, you must either truncate the result or round it up or down before stuffing it into an integer variable (assuming that you are using integer math in your software). If the accuracy of the variable or constant forces you to round to the nearest integer you would either add or subtract 0.5 from the number and then truncate it. The possibility for error arises if you're trying to round signed numbers. Let's take the following example:

```
#define
SPACE_SHUTTLE_ORBITAL_
TRAJECTORY
(SHUTTLE_CRUISE_SPEED
- (SHUT-
TLE_WEIGHT* 1.67584))
```

Realistically, the shuttle's orbital trajectory would probably be done in floating point, but stick with me here while I use the absurd to illustrate the practical. If we desired to round the number and cram it into an integer constant, the following rules would apply:

- If the number >= 0, then add 0.5 and truncate
- If the number < 0, then subtract 0.5 and truncate

Let's assume that the algorithm designer uses constants for the shuttle cruise speed and weight

Listing 4 System hardware definition file

```
/* Analog to Digital Hardware Parameters */
#define              A2D_VOLTAGE_RANGE_  (5)                 /* VOLTS */
#define              MAX                 (0)                 /* VOLTS */
#define              A2D_VOLTAGE_RANGE_  (3.0)               /* VOLTS */
/* Sensor Hardware  MIN                 (((SENSOR_ZERO_
Parameters */        A2D_RESOLUTION      VOLTAGE/(A2D_VOLT-
#define              (1024) /* 10 BITS   AGE_
#define              */
                     SENSOR_ZERO_VOLT-
                     AGE
                     SENSOR_ZERO_POINT
RANGE_MAX - A2D_VOLTAGE _RANGE_MIN))*
(A2D_RESOLUTION))
```

that result in a positive trajectory. The #define could be coded as:

```
#define \
SL_ SPACE_SHUTTLE_OR-
BITAL_TRAJEC-
TORY (signed
long)
(SHUTTLE_CRUISE_SPEED
- (SHUT-
TLE_WEIGHT* 1.67584) +
0.5)
```

Years later, the shuttle cruise speed is reduced resulting in a negative shuttle orbital trajectory constant. The rounding in this example is done incorrectly and the shuttle crashes into the earth at about 2,000 MPH. Oops! To eliminate this problem, create macros for doing the rounding. And while you're at it, you can even throw a type cast into the macro, as shown in Listing 3 .

Now when the preprocessor calculates the equation, the macro checks for the sign of the result and then adds or subtracts 0.5 as appropriate, and the government doesn't have to raise our taxes to pay for the mistake:

```
#define
SL_SPACE_SHUTTLE_ORBIT-
AL_TRAJEC-
TORY SL_RND(SHUTTLE_
CRUISE_SPEED
- (SHUTTLE_WEIGHT*
1.67584))
```

System parameters

When possible, tie your constants and parameters back to some physical unit. Then if the dimensions of your external system change, you can change the physical unit constant defined in your header file so that all constants based on the physical unit will be recalculated at compile time. Of course, you have to check for overflows, but hopefully the software is defined well and you are given limits for the physical unit.

For example, let's say we have a 10-bit analogue-to-digital voltage converter (A to D) with a full swing of zero to five volts. Let's say that the zero-point for the sensor in which we're interested is at +3.0V. You could manually calculate the zero point A-to-D value by taking 3/5 * 1023 and hard code the value 614 into the software:

```
#define SENSOR_ZERO_POINT
(614)
```

If any part of the external architecture changes (voltage range, zero point of the sensor, A-to-D resolution) the software engineer must recalculate the A-to-D zero-point value manually. A better method is to define the architectural parameters in the hardware header file, as shown in Listing 4 .

The value in which we're interested is the SENSOR_ZERO_POINT . If any part of the parameter of the system changes, the value of the SENSOR_ZERO_POINT is automatically calculated.

The same concept holds true for constants that are time-based. You should tie them back to the physical oscillator or internal clock frequency. When you change processing speeds, all you have to do is alter the clock frequency and all time-based constants are recalculated.

On the defensive

Many of the ideas I've raised are familiar to software developers, especially those with years of coding experience. Unfortunately, experience is something you don't get until just after you need it. There are many books authored on software development methods for safety-critical systems that define techniques to help reduce the occurrence of errors. An article of this length cannot do justice to the topic, but I hope your interest in defensive C coding practices has increased. For more information on the topic, check out the references below.

References

1. ANSI/ISO 9899-1990, "For Programming Languages—C," American National Standard Institute, New York, NY. Back
2. Humphrey, Watts S. A Discipline for Software Engineering . Reading, MA: Addison Wesley, 1995. Back

Other sources

Abrial, J.R. The B-Book Cambridge, U.K.: Cambridge University Press, 1996.
Douglass, Bruce P. Real-Time UML: Developing Efficient Objects for Embedded Systems Reading, MA: Addison-Wesley, 1998.
Hatton, Les. Safer C: Developing Software for High-Integrity and Safety-Critical Systems New York: McGraw Hill, 1995.
IEEE. Software Engineering Standard, 3rd Edition, New York: Institute of Electrical and Electronic Engineering, 1989.
Leveson, Nancy G. Safeware: System Safety and Computers
Neumann, Peter G. Computer Related Risks . Reading, MA: Addison-Wesley, 1995.
Spinello, Richard A. Case Studies in Information and Computer Ethics Englewood Cliffs, NJ: Prentice Hall, 1997.
Storey, Neil. Safety Critical Computer Systems Reading, MA: Addison-Wesley, 1996.
