TI | Developer Conference
March 7-9, 2007 • Dallas, TX

Minds in Motion

Writing Reliable C/C++ Systems Code

The Nuts and Bolts

Greg Davis
Technical Lead, Compilers
Director of Engineering

Green Hills®
• SOFTWARE, INC. •

Contents

Why Reliability?

- Software engineering:
  - Increasingly more expensive
  - Problems persist after last hardware kinks have been worked out.

Glitches

- Bugs that are hard to reproduce take the longest to fix.
- Often due to software bugs that fail in ways that depend on a lot of environmental factors.
- I don’t have a silver bullet for race conditions, but I can suggest things I’ve seen hit people time and time again…

Why Reliability

- Delays due to software engineering are the #1 reason that products ship late.
  - Delays in the final integration and testing stages of a product are the most costly.
- On-time products make money
- Late products make less money

Software Engineering Research

- Object-oriented programming (OOP)
  - Promises to allow code re-use by encouraging well defined interfaces and polymorphic code.
- XP
  - Encourages code re-use by testability.
  - Discourages making solutions that are overly general.

But what about coding?

Coding

- Coding technique is to a large extent independent of software design.
- Proper coding standards can make your programs:
  - More reliable
  - More efficient
  - More portable
- This talk will focus on practical, yet often overlooked, coding standards that will make your code more reliable.

Bit Sizes

- In C, types do not have guaranteed sizes.
  - Char: must be at least 8 bits, but can be larger. Most control code people expect 8 bit chars, but 16 bits is common in the DSP world. The signedness of a “plain” char also varies.
  - Short: most people expect 16 bit shorts, but they can be larger.
  - Int: most 32-bit programmers expect 32 bit ints, but they can be as little as 16 bits or even larger than 32 bits.
  - Long: not guaranteed to be 32 bits. Could be 40 bits, 64 bits, etc.

Bit Sizes

Consider:

```
int j;
for (j = 0; j < 64; j++) {
    if (arr[j] > j*1024) {
        arr[j] = 0;
    }
}
```

Type sizes

- Many embedded projects already have their own types defined.
- This is good, but they’re sometimes used loosely. The “type at hand” is often times made to do the wrong kind of job and bits get lost.
- Use run-time checks during debug mode to ferret out these problems.

“Type at hand” bug

```
typedef struct { uintptr_t sector, offset_typ off }
   head_loc;
head_loc *compare(const head_loc *a,
                                   const head_loc *b) {
{
   offset_typ t1 = a->off; l1 += a->sector;
   offset_typ t2 = b->off; l2 += b->sector;
   return (t1 > t2) ? a : b;
}
```

C99 stdint.h

- There are three variants of length-specific types:
  - Exact length: **int16_t** and **uint16_t**
  - Smallest type of at least a given length: **int_least16_t** and **uint_least16_t**.
  - The most efficient type of at least a given length: **int_fast16_t** and **uint_fast16_t**.
- Also (u)intptr_t for pointer arithmetic.

Alignment

- Data needs to be aligned. The compiler normally takes care of this automatically, but watch out:
  - When casting to a new pointer type. An int * needs more alignment than a char *.
  - When copying data using a special routine (DMA, LDM/STM copy, etc)

Alignment bug

```
// Copy data from ROM to RAM
// Show new memcpy() on the next slide that will properly check for
     misalignment
copy_data::
    ldr r0, =ghsbegin_data
    ldr r1, =ghsbegin_ROM_data
    ldr r2, =ghssize_data
    mov r2, r2, lsr 2 // scale by 4
top:
    subs r2, r2, 1
    ldrne r3, [r1], 4 // load if != 0 and postincrement by 4
    strne r3, [r0], 4 // store if != 0 and postincrement by 4
    bne top
```

Alignment Solutions

- Use assertions during debugging to verify the alignment of pointers.
- Turn on the ARM alignment checking in co-processor 15, register 1. C6x core will normally trap on misaligned loads.

Alignment Solutions

```
// ptr may be misaligned
int get(somestr *ptr) {
    // naturally aligned
    somestr tmp;
    memcpy(&tmp, ptr,
           sizeof(somestr));
    … // use tmp
}
```

```
// ARM compilers
int get_word(int *ptr)
{
    __packed int *p = ptr;
    return *p;
}
```

Alignment solutions

```
// TI C6x compiler solution
// p1 and p2 may be misaligned
void copy(int *p1, int *p2)
{
    int tmp = _mem4(*p1); // compiler intrinsic
    _mem4(*p2) = tmp;
}
```

Packing

- Structure packing is sometimes used to save space or to try to match a structure layout to hardware or a foreign system.
- Taking the address of a field of a packed structure can also result in alignment problems.
- The previous solutions work here.

Memory optimizations

- Optimize redundant accesses

```
mem = 2;      /* Redundant */        →    /* Removed: mem = 2; */
mem = 1;                                  mem = 1;
```

- Optimize load size

```
load_word  0(r3) -> r4               →    load_signed_halfword x(r3)-> r4
sign_extend_halfword r4 -> r4               where x==0 for little endian and
                                                  x==2 for big endian
```

Volatile

- Prevent memory optimizations on specific entities by declaring variables or structure fields as volatile.
  - All volatile loads and stores specified in the program will be performed.
  - The proper access size will be used (bit fields use the container size and then mask off excess bits)
- Volatile is useful for:
  - Memory mapped I/O or control registers
  - Memory shared between threads or processors
- Don’t overuse – optimization suffers!

Touching volatile memory

- Sometime necessary to read, but value is unimportant.
- Common idiom:

```
// What’s wrong?
int read2(volatile int *p)
{
    *p; // discard first read
    return *p;
}
```

Touching volatile memory

Avoid C++ pitfall!

```
#if defined(__cplusplus)
template inline void touch(T &p) { (T)(p); }
#else /* C */
#define touch(p) (void)(p)
#endif
int read2(volatile int *p)
{
    touch(*p); // discard first read
    return *p;
}
```

Minds in Motion

Violation of C Aliasing

- C and C++ have rules about what types can alias other types.
  - Use the right type or a char to access memory
  - Or use a memory barrier to prevent optimizations from doing the wrong thing
- Bad example:

```
float negate_float(float *p)
{
        *(int *)p ^= 0x80000000;
        return *p;
}
```

Assembly code

- More prevalent in embedded programming.
- Common operations include:
  - Enable/disable/restore interrupts
  - Memory barrier
  - Mutual exclusion operations
  - Operations that are not efficiently expressible in C.

Avoiding assembly

- Sometimes it is tempting just to throw in a couple of lines of straightforward assembly:

```
// Enable interrupts on ARM
asm {
    mrs                 R0,CPSR
    bic         R0,R0,0xc0
     msr        CPSR,R0
}
```

Inline Assembly Code

- Embedded assembly code is fragile and interacts with debuggers in strange ways.
  - What if compiler tries to allocate something to r0?
- When assembly code is necessary, it is good practice to isolate it using a functional interface.

Functional Interface

- C/C++ declaration:

```
extern uint32_t
    bswap32(uin32_t val);
```

```
     .text
     .globl bswap32
bswap32:
    ; R0 = A, B, C, D
EOR     R1, R0, R0, ROR #16
    ; R1 = A^C, B^D, C^A, D^B
BIC     R1, R1, #0xFF000
    ; R1 = A^C, 0, C^A, D^B
MOV     R0, R0, ROR #8
     ; R0 = D, A, B, C
EOR     R0, R0, R1, LSR #8
     ; R0 =D^0,A^(A^C),
            B^0,C^(C^A)
     ; R0 = D, C, B, A
BX      LR
```

Inline Assembly Code

```
INLINE uint32_t bswap32(uint32_t val) {
    uint32_t int ret;
    asm {
            EOR  R1, val, val, ROR #16
            BIC  R1, R1, #0xFF0000
            MOV  ret, val, ROR #8
            EOR  ret, ret, R1, LSR #8
      }
      return ret;
}
```

Avoiding assembly

- Can sometimes even re-code to avoid assembly:

```
#define ROR32(x, y) (((x) >> (y)) | (x << (32 – y)))
INLINE uint32_t bswap32(uint32_t val)
{
    uint32_t tmp1, tmp2;
     tmp1 = val ^ ROR32(val, 16);
     tmp1 &= ~0xFF0000;
     tmp2 = ROR32(val, 8);
     return tmp2 ^ (tmp1 >> 8);
}
```

Avoiding assembly

- Many toolchains even make it easier by providing common operations as intrinsic functions. These look like function calls, but they are translated into assembly instructions, usually in a way that the compiler understands and can optimize.

```
__EIR(); // ARM intrinsic
_enable_interrupts(); // C6x intrinsic
```

What’s wrong?

```
uint32_t unpack(uint8_t *p) {
    uint32_t acc, i, err = 0;
    for (i = 0; i < 4; i++)
        acc = (acc << 8) | p[i];
    if (acc == 0xdeadbeef)
        err = 1;
    if (err == 1)
        panic(“bad value”);
    return acc;
}
```

MISRA C

- Designed for the automotive market
- 141 rules that define a subset of full C
- Remove:
  - Non-portable constructs (e.g. sizes of types)
  - Easily confused things (octal constants, side effects in big expressions, etc)
- Various checkers for MISRA C are available

TI | Developer Conference
March 7-9, 2007 • Dallas, TX

Writing Reliable C/C++ Systems Code

Greg Davis
gdavis@ghs.com

Minds in Motion

Guidelines for Writing Reliable C/C++ Code

By Greg Davis
Technical Lead, Compiler Development
Green Hills Software, Inc.

Introduction

Simple coding techniques can make your embedded system more reliable.

Does this sound too good to be true? Software engineering has become the most expensive aspect of many kinds of embedded product development. Software is also one of the more problematic aspects as software bugs plague products long after the last hardware problems have been resolved or worked around. Thus, it is not surprising that much has been written about improving software engineering practices. Much of this has to do with the practice of how to design, program, and test your programs in the most efficient manner.

Regardless of the software engineering methodology employed, there are low level decisions that are usually left up to the individual developer. Although most organizations have some kind of coding guidelines, these often focus on issues like naming variables and indentation style. This paper will suggest some low-level coding guidelines that are applicable to almost all approaches to software engineering. While no one can promise you perfectly reliable codes, these steps will surely take you in the right direction.

Bit Sizes

Many people forget that the C and C++ languages do not define the sizes of various types. For example, a "char" must be at least 8 bits long, but it can be any length at all. In fact, C defines the minimum sizes for all the different built-in types, but the exact sizes are left up to the implementation.

Programs can implicitly depend on the size of a type. For example:

```
int j;
for (j = 0; j < 64; j++) {
    if (arr[j] > j*1024) {
        arr[j] = 0;
    }
}
```

On a target where an int is a 16-bit quantity, **j*1024** will overflow and become a negative number when **j >= 32**.

Many embedded projects have already standardized on a system of typedefs that define types of known lengths for given tasks. While this is great, it is often times not enough. All too often, the programmer will misuse a type that he's used to for something that it is not appropriate for. For example, I occasionally look at customer-reported bugs that look something like this:

```
array_offset_t off;
... // sets off to an offset within array “arr”
off += (array_offset_t) arr;
record_offset((address_t)off);
```

This is erroneous, because it is quite possible for array_offset_t to be a 16-bit type on a resource-constrained embedded system with a 32-bit address_t type. The “off” variable may overflow when the user attempts to add the address of array to it. The user, misunderstanding the type structure, peppered the code with casts to make the lint-style checks, but he managed to miss the point of the type system. Run-time error checking or sophisticated static analysis may help to catch some of these bugs, but nothing is a substitute for what conscious programming.

If you are not currently using a size-independent type system, I might suggest looking at the approach defined in the 1999 C specification. The **stdint.h** header file defines three variants of the C99 length-specific types:

1. Types that are exactly equal to a certain width. These types are given names like **int16_t** and **uint16_t**.
2. The smallest type that has at least a certain width. These types have names such as **int_least16_t** and **uint_least16_t**.
3. The most efficient type that has at least a certain width. These types have names such as **int_fast16_t** and **uint_fast16_t**.

The first form is probably mostly necessary in the case where the type is expected to truncate or wrap around after reaching the maximum value. Note that there is no guarantee that the first form will exist for a given size. Code that depends on this type may need to be manually transformed if you are moving to a machine that doesn't support this size. Most common processors support 8, 16, and 32-bit types, while compilers for 32-bit processors and larger generally support a 64-bit integral type with reasonable efficiency. The second form is good for structure definitions where the programmer is attempting to save space, yet needs a guaranteed amount of storage precision. The third form is best for computational variables (particularly automatic variables) where the user needs at least a certain number of bits of precision.

Using the right **stdint.h** types does not eliminate all portability problems based on the sizes of various types[1], but it eliminates most of them.

Alignment

Data needs to be aligned based on the processor's alignment constraints when accessing memory. The compiler normally takes care of this automatically, but watch out for the following cases:

1. When casting to a new pointer type. An int * needs more alignment than a char *.
2. When copying data using a special routine (DMA, LDM/STM copy, etc)
3. When using packed data, (e.g. #pragma pack, __attribute__((packed)), etc.)

Whenever using such data, it is best to ensure that it is aligned. Assertions may be used to look for pointer casts that are not appropriately aligned. When potentially unaligned data must be accessed, one of a few methods may be used:

1. Some compilers support a keyword, such as __packed, for referencing data through unaligned pointers.
2. Use macros for accessing the data. Define the macros to access the data in smaller chunks on systems where this is necessary.
3. The data may be copied to an aligned buffer, using a memory routine that gracefully handles unaligned data, such as memcpy. For example:

[1] The "integral promotion" rule states that before chars and shorts are operated on, they are cast up to an integer if an integer can represent all the values of the original type. Otherwise, they are cast up to an unsigned integer. The following code behaves differently on a target with a 16-bit integer (where it returns 0) than it will on a target with a 32-bit integer (where it returns 65536).

```
int32_t a()
{
   int16_t x = 65535;
   int16_t y = 1;
   return x+y;
}
```

```
// somestr is a normally aligned structure, but
// upon entry to this function, ptr may be
// misaligned as it was cast from a byte buffer.
int get(somestr *ptr) {
    // Allocate a naturally aligned
   somestr tmp;
   memcpy(&tmp, ptr, sizeof(somestr));
   // use tmp
}
```

Memory Optimizations

Compilers assume that most memory behaves in a straightforward manner where the value read is the same as the last value written. This allows a number of optimizations to be performed.

Example 1: optimize redundant access.

Code before	**Code after**
mem = 1;	/* dead store optimized away */
mem = 2;	mem = 2;

Example 2: optimize load size

Code before	**Code after**
load_word 0(r3) -> r4	load_signed_halfword x(r3) -> r4 where x==0 for little endian and x==2 for big endian
sign_extend_halfword r4 -> r4	/* optimized away */

These optimizations are powerful and greatly improve the quality of the code that the compiler generates, but they can disrupt a working system in a number of cases:

1. When the memory is shared between different threads in a single memory space, one thread may sit in a busy loop, reading a variable over and over again, and waiting for its value to be set by another thread. For example:

```
extern int value;
while (value == 1)
    (void)0; // wait for value to be cleared
…
```

Obviously, the task could hang if this were optimized into the apparently equivalent sequence:

```
temp = value;
while (temp == 1)
    (void)0;
```

2. When the memory maps to a hardware device where reads and writes may have side effects.

The solution to these problems is to use the `volatile` keyword for variables or fields of structures that should not have memory optimizations performed on them. Using the `volatile` keyword should ensure:

1. That all loads and stores are performed as expressed in the original source code.
2. That the entire declared size of the object is used for all accesses.

Many problems related to memory optimizations manifest themselves as race conditions that can be very difficult to debug without advanced debugging tools. As a result, the `volatile` keyword should be used proactively when writing code that deals with hardware devices or shared memory. However, don't use `volatile` unnecessarily; it inhibits powerful optimizations that are safe in most cases.

As a final note on the topic of using volatile, it is sometimes necessary to read a volatile hardware register even though the value itself is not necessary. For example, you may wish to throw away a header on a packet transmitted over a serial device like a UART. The normal idiom is to do something like:

(void)UART->RBR; // Read next byte

While this works perfectly well in C, through a conversion process known as “lvalue to rvalue”. In C++, however, this lvalue to rvalue conversion is suppressed on simple “expression statements” like in the above example. The safe, and clear, way to write this is to use a macro.

```
#ifdef __EMBEDDED_CXX
// Embedded C++ does not allow templates.
inline void touch(volatile int &p) { (int)p; }
inline void touch(volatile unsigned int &p) { (unsigned int)p; }
// repeat for other types as necessary...
#elif defined(__cplusplus)
template  inline void touch(T &p) { (T)(p); }
#else /* C */
#define touch(p) (void)(p)
#endif
...
    touch(UART->RBR);
```

C Aliasing Rules

C and C++ have rules about what types can alias other types. These rules basically state that only the declared type [2]of an object[3] or a character may be used to reference an object. So, the following code is illegal:

```
float negate_float(float *p)
{
  *(int *)p ^= 0x80000000; // int may not alias float!
  return *p;
}
```

While you will usually get away with code like this, you can run into trouble in more complicated cases. For example:

```
float negate_float(float *p, int *p2) // p and p2 alias
```

[2] The standard uses the term “effective type” which is somewhat more involved than this. While, in my opinion, there are some obvious holes in the standard’s treatment of the subject, its intent is clear

[3] An object is, roughly speaking, a variable in C or C++. It is not necessarily an object in the object-oriented sense.

```
{
   *p2 ^= 0x80000000; // modifies *p
   return *p;
}
```

The compiler may hoist the load of `*p` into or above the previous statement, which may result in the non-negated value of *p being returned.

Assembly Statements

Embedded applications are more closely tied to their underlying hardware, so assembly code is much more common than in desktop applications. Because the assembly languages vary from one machine to another, assembly code is non-portable. More importantly, it strays beyond the definition of the language, and, as a result, its behavior is often not well defined.

Whenever possible, break assembly code apart into separate functions that are callable by the standard C interface so that assembly code does not need to be mixed into the middle of C code. This will help keep the non-portable parts of the program separate, which helps when it comes time to port the program to a new system.

For example, you might want a routine to swap the bytes in a word. This routine, **bswap32()** can be declared in a header file as:

```
extern uint32_t bswap32(uin32_t val);
```

Then this could be defined in an assembly file. Let us suppose that we are programming for the ARM architecture. This could be implemented in an assembly file called cpu.s using the following tuned sequence of instructions:[4]

```
        .text
        .globl bswap32
        ;  extern uint32_t bswap32(uin32_t val);
bswap32:                                ; R0 = A, B, C, D
```

[4] David Seal, editor, **ARM Architecture Reference Manual**, copyright 1996-2000 by ARM Limited.

```
        EOR   R1, R0, R0, ROR #16              ; R1 = A^C, B^D, C^A, D^B
        BIC   R1, R1, #0xFF0000                ; R1 = A^C, 0, C^A, D^B
        MOV   R0, R0, ROR #8                   ; R0 = D, A, B, C
        EOR   R0, R0, R1, LSR #8               ; R0 = D^0, A^(A^C), B^0, C^(C^A)
                                               ; R0 = D, C, B, A
        BX    LR                               ; return.
```

This will do well from a functional point of view. However the byte swap functionality may be needed in performance critical parts of the code that cannot afford the overhead of a pair of call and return instructions.

An inlined function might do well in a case like this. One could re-write this code as:

```
INLINE uint32_t bswap32(uint32_t val)
{
   uint32_t int ret;
   asm {
        EOR   R1, val, val, ROR #16
        BIC   R1, R1, #0xFF0000
        MOV   ret, val, ROR #8
        EOR   ret, ret, R1, LSR #8
   }
   return ret;
}
```

This sort of inlined assembly approach is popular as it isolates the assembly language to a header file and it offers the hope of efficient code. Still, it suffers from the same reliability problems as any other assembly code, although, due to being defined a single time, there is less of it spread throughout the code.

An even better approach that is portable and will generate the same code is to use portable C:

```
#define ROR32(x, y) (((x) >> (y)) | (x << (32 – y)))
INLINE uint32_t bswap32(uint32_t val)
{
   uint32_t tmp1, tmp2;              /* val = A, B, C, D */
   tmp1 = val ^ ROR(val, 16);        /* tmp1 = A^C, B^D, C^A, D^B */
   tmp1 &= ~0xFF0000;                /* tmp1 = A^C, 0, C^A, D^B */
```

```
    tmp2 = ROR(val, 8);              /* tmp2 = D, A, B, C */
    return tmp2 ^ (tmp1 >> 8);       /* return D^0, A^(A^C), B^(0), C^(C^A) */
                                     /*     ==  D, C, B, A */
}
```

This maintains the performance advantage of the assembly code, but is completely portable. Semantically, it behaves like a function call, but it does not have the overhead of a function call. When it comes time to port the application to a new architecture, **bswap32()** will continue to work properly. Eventually, **bswap32()** could be re-tuned for the new architecture.

Another common task that is often done in assembly is to re-enable interrupts. For the ARM, this can be done in three simple instructions:

```
        mrs     R0,CPSR
        bic     R0,R0,192
        msr     CPSR,R0
```

Given the simplicity of this sequence, one might be tempted to inline these instructions into the code whenever they are needed by doing something like:

```
asm {
        mrs     R0,CPSR
        bic     R0,R0,192
        msr     CPSR,R0
}
```

Once again, this is a mistake. It is best to keep the interface as portable as possible by using an inlined function:

```
 INLINE void EI(void);
```

Some toolchains allow the implementation of assembly instructions by intrinsic functions. They look like function calls, but they are translated into assembly instructions. Their semantics, however, behave like real function calls; in essence, their translation into specific inline assembly instructions in an optimization.

These are also inherently portable since the intrinsic functions can be re-defined on a new architecture with an equivalent sequence of instructions. Some standards even exist, such as the PowerPC AltiVec Programming Interface and the ETSI intrinsics, which define common interfaces across different compilers. Intrinsics are becoming the standard, while inline assembly is increasingly seen as a hindrance to portable code.

Safer C is Reliable C

While the C language is the most popular language for 32-bit embedded systems, by far, it has a number of pitfalls that can confuse even the most experienced software developers. While there are alternative programming languages, many of them are proprietary and would lock the developer into a particular vendor.

It would be nice to allow developers to use C, if it could be made safe. One attempt is worth mentioning here.

MISRA C was originally published in 1998 under the name Guidelines for the use of the C language in vehicle based software. The original standard consisted of only 127 rules that defined a subset of the C language. In the time since the original publication, there have been some questions and concerns about some of the rules which motivated the second edition of the standard (MISRA 2004). MISRA 2004 contains a number of clarifications and improvements to the original rules.

MISRA C is designed for the safety critical market. It prohibits the use of memory allocation[5], for example. As a result, it can be a difficult to totally comply with. However, many of the rules are universally applicable. A number of commercially available checkers may be used to verify that your code complies with selected MISRA C rules, making an incremental adoption possible. MISRA C, in its entirety, is obviously not for everyone, but anyone who writes C code will find parts of MISRA's coding guidelines useful.

Conclusion

We live and work in an increasingly complicated world where we have to juggle many roles as once. As developers, we aim to write codes that work well, but many of us are also under great time pressure. While reliability often takes a backseat to other concerns, the reliability of your embedded system directly affect your product's success in the marketplace as well as your company's reputation. While there are no silver bullets that will make your product totally reliable, adherence to a number of common sense rules are a step in the right direction.

[5] MISRA C allows for deviations to the rules under certain cases.