Automotive Embedded C Techniques and Principles

Automotive Embedded C Techniques and Principles: Key Assumptions

- Basic understanding of C.
- Basic understanding of embedded systems.
- Focus on embedded systems.
- More than one approach may be correct.
- Not a comprehensive list.

Automotive Embedded C Techniques and Principles: Definitions

Software Anomaly: any unexpected behavior exhibited by the software regardless of whether the anomaly is induced by external module conditions (for example an electrical transient or a gamma ray that flips a bit in a control register) or internally caused by a hardware anomaly (for example, the watchdog timer fails to initialize) or internally caused by a software problem (for example, a stack overflow not detected during software testing).

Robustness: a quality measure of the software's ability to perform all intended physical or functional operations within a known, short time duration regardless of all possible module input and environmental conditions including processor load, frequency of interrupts, frequency of vehicle network communication rates. Embedded software is not robust when the software design and implementation have not addressed all possible conditions that might introduce a **Software Anomaly.**

Efficiency: a measure of the software's use of critical computer resources. For our purposes this means program memory (ROM, FLASH), data memory (RAM, EEPROM), and CPU execution time.

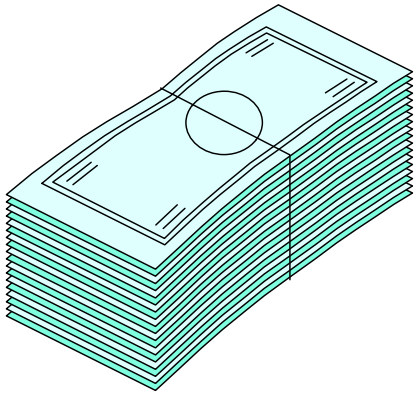
Automotive Embedded C Techniques and Principles: Content

- **Robust C Techniques and Principles**
  - Some motivation
  - Techniques
  - Principles
- **Efficient C Techniques and Principles**
  - Some motivation
  - Principles
  - Techniques

Motivation

Risks

- Software Bugs may have dramatic consequences
- Software Bugs are systematic i.e. in every car!
- Supplier pays for recall costs
- Product Liability
- SW complexity often underestimated

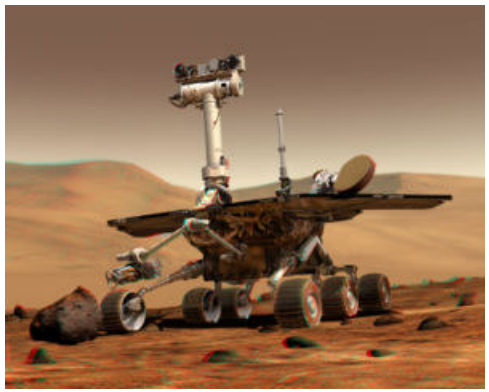


Motivation

Famous SW Bugs

- NASA Apollo 11 landing problem (July 20, 1969).
- ESA Ariane 5 Flight 501 self-destruction 40 seconds after takeoff (June 4, 1996).
- NASA Mars Climate Orbiter destroyed due to entry of momentum data in imperial units instead of the metric system (September 23, 1999).

- NASA Mars Rover freezes due to too many open files in flash memory (January 21, 2004).
- NASA Mars Global Surveyor battery failure was the result of a series of events linked to a computer error made five months before (November 2, 2006).



Motivation

Famous SW Bugs

- The Therac-25 accidents (1985-1987), which caused at least five deaths.

- A misuse of medical diagnosis software created by Multidata Systems International, at the National Cancer Society in Panama City, caused, by different estimates, between five and eight cancer patients to die of over-radiation. (2000)

- The year 2000 problem, popularly known as the "Y2K bug", spawned fears of worldwide economic collapse and an industry of consultants providing last-minute fixes.

- The Pentium FDIV bug.

Motivation

Famous SW Bugs

- The 2003 North America blackout was triggered by a local outage that went undetected due to a race condition in General Electric Energy's XA/21 monitoring software.

- The software error of a MIM-104 Patriot, which ultimately contributed to the deaths of 28 Americans in Dhahran, Saudi Arabia (February 25, 1991).

- Chinook crash on Mull of Kintyre: the cause of this event remains a mystery, but strong suspicions have been raised that software problems were a contributory factor.

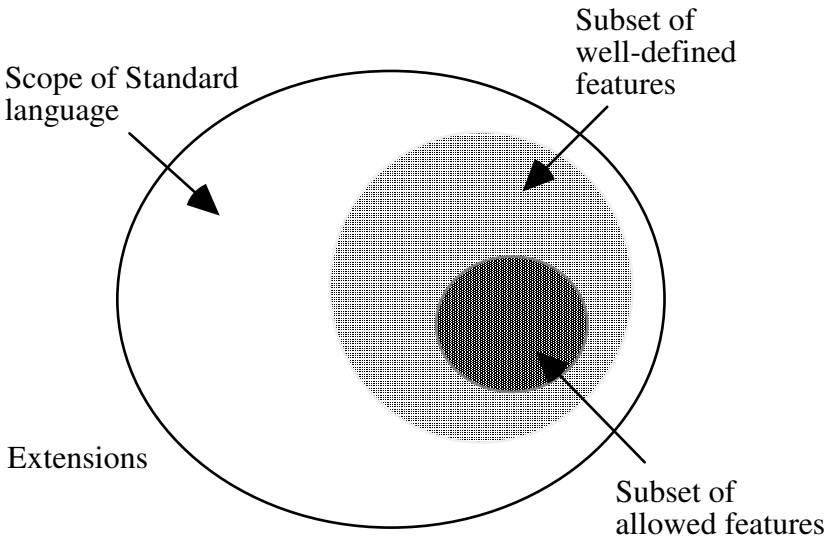
Robust C: Lessons to Learn

Software engineers too often believe defects can be eliminated and do not plan for their inevitable appearance. Some lessons:-

- Software failure is expensive and very unpredictable
- a quick-fix can never be justified
- diagnostic hooks are very inadequate in released systems
- repetitive failure dominates software systems
- Many software failures could have been avoided using techniques we already know how to do.

Robust C: Define a Safe(r) Subset of the C Language

The subset nature of programming languages



EXTENSIONS - MICROCONTROLLER COMPILER SPECIFIC NOT PORTABLE

SUBSET OF WELL DEFINED FEATURES - ANSI C

SUBSET OF ALLOWED FEATURES - MISRA CARBODY C CODING GUIDELINES WHAT NOT TO USE

Robust C: Use of Micro Specific C Extensions

In order to facilitate the portability of C programs, a general rule of thumb is to use ANSI C constructs as much as possible, especially function prototyping. For cases that require tighter code, use C compiler's language extensions – to access hardware, locate variables in memory, specify interrupt service routines, etc. The disadvantage is that using language extensions renders the C program non-portable.

Robust C: Automotive C Standard - MISRA Guidelines

In April 1998, the ***Motor Industry Reliability Association*** (**MISRA**) published a set of C guidelines for use in vehicle-based software. www.misra.org.uk

MISRA C Guidelines

MISRA is the primary recommended practice for embedded C in the automotive industry. It promotes many best practices.

Use a MISRA compliance checking tool during development before you run the compiler. Performing such a check is called *Static Analysis*.

MISRA compliance is much easier to meet if you keep it in mind as you write code. It is much more difficult to take existing code and make it MISRA compliant later.

Robust C: Coding Technique - Function Prototypes

Technique: Always use function prototypes.

Justification: MISRA Rule 71 (required). Function calling affects the stack useage (depending on parameters passed). The compiler needs to know how much stack space needs to be reserved for every function call. If no prototype is given then the ANSI C default function prototype is used. This may increase the stack size.

int foo (int);

Exceptions: none.

Robust C: Coding Technique - Side Effects

Technique: No conditional statement shall contain side effect, e.g. assignments, bit shifts, etc. inside IF, CASE or loop controls.

Justification: Reduces the potential for unintended side effects. Increases software readability and maintainability.

Exceptions: Initial assignment on iterative loop controls.

Robust C: Coding Technique - Assembly Useage

Technique: Use of Assembly language shall be limited to hardware access. Assembly language shall be isolated in a separate module, subroutine or macro.

Justification: MISRA Rule 3 (advisory). Prevents the compiler from performing optimizations. Makes the code not portable.

Exceptions: For reasons of efficiency it is sometimes necessary to embed simple assembly language instructions in-line, for example to enable and disable interrupts, initialization, etc. If it is necessary to do this for any reason, the it is recommended that it be achieved by encapsulating the assembly in a macro.

Robust C: Coding Technique - Variable Declarations

Technique: Variables are to be declared with the smallest possible scope. Function scope (local variable) or file scope should be used. Naming convention should reflect scope of variables.

Justification: MISRA Rule 22 (advisory). Using the smallest possible scope improves the code readability, reduces risk of undesired use of a variable or name conflict.

Exceptions: Procedure-oriented nature of C does not allow consistent use of this requirement. The idea of limited scope for data found its true implementation only in an object-oriented language as C++. C++ allows data encapsulation.

Robust C: Coding Technique - Numeric Constants

Technique: All numeric constants in software must be mapped to a symbolic name, which is defined once in one location only.

Justification: Standard good programming practice. Protects against changes to these values. Guarantees that when a constant changes, it changes everywhere in the code.

Exceptions:

- The constant number zero may be used without defining it as a symbolic constant.
- Numeric constants may also be used in array/matrix/structure initialization where a symbolic constant does not exist. However, an explanation of the origin of the constants must be included in the comments.

Robust C: Coding Technique - Volatile Usage

Technique: Define all shared resources as volatile.

Justification: Carbody C Coding Rule 132 (required). Avoids unexpected optimizations which may create mutual exclusion problems.

Exceptions: none.

Robust C: Coding Technique - ISR Global Variable Access

Technique: Take extreme care in accessing global variables within ISR's!

- Determine the direction of data flow (task->ISR or ISR->task).
- Use the data flow direction to determine who reads the variable only and who writes to it.
- Use an atomic statement (uninterruptable statement) when you read or write the variable in order to guarantee mutual exclusion. If necessary disable interrupts temporarily.
- Only read a ISR global variable once during a task cycle so that its value does not change. If necessary, make a copy into a local variable.
- Make a code review on sections of code which access ISR global variables.

Justification:

- An interrupt can happen at any time!
- Must guarantee the mutual exclusion of global variables!

Exceptions: none.

Robust C: Coding Technique - Refresh of State/Port Values

Technique: Refresh the values of state variables and output ports constantly. Do not assume that once set that it is stable.

- **E.g. on event E1, switch ON the motor output P1 for a time T1**
  - When in the idle state, set the motor P1=OFF all the time.
  - On event E1, start timer T1.
  - If the timer T1 is running, set motor P1=ON all the time.

Justification:

- Port registers and RAM values may be corrupted by EMC events or other code.

Exceptions: In some cases there is a tradeoff between robustness and code efficiency. In such cases, this code should be reviewed!

Robust C: Coding Technique - Multi-Purpose Global Variables

Technique: Do not create multi-purpose global variables!

Justification:

- The possibility for misuse is very high!

Exceptions: None.

Robust C: Coding Technique - Data Types

Technique: Use Project type definitions *instead* of basic C types (ie., char, int, short, long, etc.)

Justification: MISRA Rule 13 (advisory). The storage of length types can vary from compiler to compiler. It is safer if programmers work with types which they know to be a given length.

Exceptions: None.

Type	Definition	Size	Range
T_UBYTE	unsigned byte	1	0..255
T_UWORD	unsigned word	2	0..65535
T_ULONG	unsigned double word	4	0..FFFFFFFFh
T_SBYTE	signed byte	1	-128..127
T_SWORD	signed word	2	-32768..32767
T_SLONG	signed double word	4	80000000h...7FFFFFFFh
T_FLAG8	bit	1	0..1
T_FLAG16	bit	2	0..1

Robust C: Coding Principle - Variable/Function Naming

Principle: Use correct, consistent, readable names for functions and variables. Take into account:

- Name the function according to its purpose (verb + subject/object).
  - **E.g. GetID(…), CalculateResponse(…), TreatInputRequests(…)**
- Name the variable so one can easily recognize the information contained.
- Use a consistent “language” for your variables and functions.
  - **E.g. Do not call a response “response” in one place and “answer” in another.**

Justification:

- Makes the code easier to read for you and more importantly others!
- Makes maintainability of the code easier.

Robust C: Coding Principle - Expect Resets

Principle: Write code so that after a reset, the system is in a consistent state.

Justification:

- A reset can occur at any time!

Robust C: Coding Principle - Minimize IF Statements

Principle: Reduce the number of IF statements whenever possible. Attempt to design code that has fewer decisions.

Justification:

- Making a decision is a chance to make a wrong decision and to miss a possible case.

Robust C: Coding Principle - Clever Code

Principle: Don't write clever code!

Justification:

- Fewer source characters does not equal better code!
- Easier for compilers and other SWDs to read and maintain code.
- Safer for portability.
- Common C constructions are less likely to contain bugs.

Robust C: Coding Principle - Defensive Programming

Principle: Practice defensive programming!

Justification:

- Never trust anything. If it is possible to make an assertion, make it.
- If a C function returns an error status, check it.
- Make sure all logical structures are complete (default in switch, else in if .. else if).

Robust C: Coding Principle - Code Simplicity

Principle: Write code in the most straightforward manner possible. Worry about whether the next programmer (or you, after 6 months) will understand it. Let the compiler worry about making it efficient (especially for 16-bit processors).

Justification:

- Code maintainability.

Robust C: Coding Principle - Test Early and Often

Principle: Test your code early and often. Do not write 100 lines of code before putting it on an emulator!

Justification:

- There are statistics that state that in 1000 lines of new code, there are usually 60-70 mistakes. On the average, your 100 lines will have 6 or 7 mistakes! It's much easier if you write only about 15 lines before testing. Then, on average, there will be about 1 mistake and you only have to look in 15 lines for it!

Robust C: Coding Principle - Code Deliberately

Principle: Know what the code you are about to write is supposed to do before you write it!

Justification:

- You will write better code!

Robust C: Coding Principle - Assumptions

Principle: Learn to challenge your own assumptions! Even if you think you know how something works (ie., an external chip, or the CPU), review its specification for relevant points when writing the code for it or debugging it.

Justification:

- Some experience has shown that about 50% of root causes found during debugging are due to an invalid assumption or incorrectly remembering a specification.

Robust C: Coding Principle - Code Optimization

Principle: Don't optimize the code after it has been written. Try to optimize the code in the first place!

Justification:

- Optimization is another chance to make mistakes.

Efficient C: Some Motivation - PC vs. Embedded System

Non-issue for most PC based C development

- “Unlimited” computer resources (ie., memory and CPU loading).
- Labor is expensive.

Issue for embedded system based C development

- Limited microcontroller resources (ie., memory and CPU loading).
- Performance is time-critical (real-time behavior).

Efficient C: Some Motivation - Cost!

The total cost of a PC based software is only the labor cost of developing the program. The user typically already has a PC.

The total cost of a embedded system software is the labor cost of developing the software plus the cost of the hardware itself.

Example: Automotive Body Control Module (BCM)

- **Cost of Software Labor = $500.000**
- **Cost of Microcontroller = $5**
- **BCM Volume per Year = 500.000**
- **Product Life of BCM = 3 years**
- **Total Microcontroller Costs = $5 * 3 * 500.000 = $7.500.000**
- **Cost savings if software development is reduced by 50% = $250.000**
- **Cost savings if a microcontroller costing 50% less is used = $2.5 * 3 * 500.000 = $3.750.000!**

Efficient C: Tradeoffs

Tradeoffs:

- Size (Memory) vs. Speed (CPU execution)
- Efficiency vs. Readability

Efficient C: Rules Before you Optimize

Rule #1: Prove you need to optimize before you start doing it!

Rule #2: Prove where you need to optimize first!

Rule #3: Do not optimize for speed until you have profiled your code. Most of the time, you will be wrong about where your code spends most of its time and your effort will be mostly wasted!

Efficient C: General Optimizing Principles

1. Know your microcontroller!
2. Know your compiler!

Each target processor and compiler have different strengths and weaknesses. Understanding them is critical to successful software optimization. The optimizing techniques presented are general and are compiler dependent.

3. Examine the assembly output of your compiled code!

To compare the efficiency of one C language implementation versus another - compare the assembly language output. You do not have to be any assembly expert! However, you should know enough to be able to recognize statements that are essentially function calls (ie., JSR, BSR, CALL …).

Efficient C: Optimizing Technique - Array Access

Technique: Array Access - Hard-coded

Advantages:
- Address resolved at compile time.
- Fast execution.

Disadvantages:
- Uses a lot of program memory (exception: small sequences).

```
AnalogValues [ 0 ] = 12 * UNIT_VOLTS;
AnalogValues [ 1 ] = 0 * UNIT_AMPS;
AnalogValues [ 2 ] = 77 * UNIT_DEGREES_F;
```

Efficient C: Optimizing Technique - Array Access

Technique: Array Access - Looped Index

Advantages:
- Easy to read.
- Program memory efficient.

Disadvantages:
- Address calculation overhead.
- Loop processing overhead.
- Slow execution time.

```
for ( index = 0; index < 5; index++ )
AnalogValues [ index ] = 0;
```

Efficient C: Optimizing Technique - Array Access

Technique: Array Access - Incremental Index

Advantages:
- Fast (but not as fast as hard-coded)
- More flexible than hard-coded.

Disadvantages:
- Uses a lot of program memory.

```
index = 0;
AnalogValues [ index++ ] = 12 * UNIT_VOLTS;
AnalogValues [ index++ ] = 0 * UNIT_AMPS;
AnalogValues [ index++ ] = 77 * UNIT_DEGREES_F;
```

Efficient C: Optimizing Technique - Binary Math

Technique: Use binary math when possible.

- Shift-left (<<) for multiply by two
- Shift-right (>>) for divide by two

Advantages:

- Microprocessor only works with in binary
- Binary math is very fast
- Binary math is memory efficient

```
i *= 4;
movlw ((04h))
movwf btemp+1
movf (((_i))^0x80),w
fcall lbmul
movwf (((_i))^0x80)
```

```
i <<= 2;
rlf (((_i))^0x80)
rlf (((_i))^0x80)
```

Both methods multiply *i* by four. The conventional method, on the right, uses more than twice as many instructions plus a library call to multiply.

Efficient C: Optimizing Technique - Macro Functions

Technique: Macros can sometime be used instead of C functions.

Advantages:
- No function call overhead.
- Ideal for small functions.
- Ideal for functions called inside a loop. Improves readability.

Disadvantages:
- Uses a lot of program memory.

```
#define Multiply(x,y) ( (x)*(y) )
for ( unsigned char i = 0; i < 25; i++ )
Value += Multiply ( Value, i );
```

Caution: Remember the parenthesis!

Efficient C: Optimizing Technique - Static Local Variables

Technique: Use static local variables whenever possible! If a variable is only to be used by functions within the same file then use static.

- Global variables are “bad”.
- Local variables are good.
- Static local variables are really good!

Advantages:

- Fast access.
- Encapsulation (reduce side effects).
- No stack overhead.
- Variable stored in fixed RAM location.
- MISRA Rule 23 (advisory).

Disadvantages:

- Increased RAM requirements.
- May interfere with the compiler’s optimizer.

Efficient C: Optimizing Technique - Table Lookup

Technique: Use a lookup table with pre-calculated values instead of complex math calculations. You can also perform interpolation between points in a table.

Advantages:

- Very fast!

Disadvantages:

- Uses a lot of program memory.

Efficient C: Optimizing Technique - Bit Flag Toggling

Technique: Use bit flag operators and NOT logical operators on bit fields.

Advantages:

- Fast on microcontrollers with bit instructions.
- Memory efficient.

```
BitFlags.bit3 = !BitFlags.bit3;
    movlw 0
    btfss (((_BitFlags))),3
    movlw 1
    movwf btemp
    rlf btemp
    rlf btemp
    rlf btemp
    movf (((_BitFlags))),w
    xorwf btemp,w
    andlw not (((1<<1)-1)<<3)
    xorwf btemp,w
    movwf (((_BitFlags)))
```

```
BitFlags.bit3 ^= 1;
    movlw 1<<3
    xorwf (((_BitFlags)))
```

Efficient C: Optimizing Technique - Data Types

Technique: Avoid Operations Involving Different Data Types

Advantages: Processor works best with one data type.

The C compiler follows the ANSI C data type promotion rules to process expression that involve different data types, resulting in extra object code and execution time. In expressions that contain char and int, char gets promoted to int. In expressions that contain both signed and unsigned integers, signed integers are promoted to unsigned integer. In expressions that contain floating point types, float gets promoted to double.

```
char c1;
int i1, i2, i3;

fcn1() {
  i1 = c1 + i3;
}
fcn2() {
  i1 = i2 + i3;
}
```

```
_fcn1:
  ld    A,(_c1)
  test  A.7
  subb  W,W
  add   WA,(_i3)
  ld    (_i1),WA
  ret
```

```
_fcn1:
  ld    WA,(_i2)
  add   WA,(_i3)
  ld    (_i1),WA
  ret
```

Data type conversion results in extra machine code.

Efficient C: Optimizing Technique - Memory Spaces

Technique: Many microcontrollers have more than one memory space. For example, a memory space may be accessible with an 8-bit offset (page 0), another memory space requires a 16-bit offset, still some memory space requires an address space modifier. You can decrease program size by explicitly locating the frequently used variables into the memory space that requires the minimum number of bytes for addressing.

Advantages:

- Program memory efficient.

```
/* Default memory area */
int a0=0, a1;
/* Tiny memory area (page 0) */
int _tiny at0=0, _tiny at1;

void fcn (void) {
          a1 = a0;
   at1 = at0;
}
```

```
Opcode          _fcn:
E1000048        ld WA,(_a0)
F1000068        ld (_a1),WA
E00048          ld WA,(_at0)
F00068          ld (_at1),WA
FA              ret
```

Caution!: this technique may reduce the portability of the code. If too many re-used functions want to put their variables in page 0 , the application can run out of page 0 space!

Efficient C: Optimizing Technique - Repetitive Code

Technique: Find sections of code that are alike or repetitive and commonize them into a function.

Advantages:

- Reduces the size of the code.
- Makes maintenance easier.

Efficient C: Optimizing Technique - Execution Time

Technique: Do not worry too much about the average execution time. Worry instead about the worst case. As a consequence, do not put in decision “filters” to eliminate easy cases early. This has the affect to add overhead and make the worst case execution worse! Instead let the more thorough method deal with all cases.

Example: Don’t throw out serial bus messages based on partial ID information. Sometimes, you will have a long string of messages that will pass this early filter, and be handled by the later processing function anyway. All this technique does is add execution overhead.

Efficient C: Optimizing Technique - Compiler Optimizations

Technique: The C compiler may provide multiple levels of optimization. For examples, optimization levels 0, 1, 2 and 3. Usually, the higher the level, the more optimization methods will be used by the C compiler to generate machine code. However, depending on the coding style, it is possible that the size of generated code of level 3 is larger than that of level 0.

Level	Function
0	**Minimum optimization (default)**
	Stack release absorption. Branch instruction optimization.
	Deletion of unnecessary instructions
1	**Basic block optimization**
	Propagation of copying restricted ranges.
	Gathering of common partial expressions in restricted ranges.
2	**Optimization of more than basic blocks**
	Propagation of copying whole functions.
	Gathering of common partial expressions of whole functions
3	**Maximum optimization**
	Loop optimization and other miscellaneous optimization

Efficient C: Carbody Code Implementation Rules

The efficiency of the presented techniques is compiler dependent. Please take this into consideration when applying ANY of these techniques.