Efficient Embedded Programming

Shawn A. Prestridge

Senior Field Applications Engineer, IAR Systems

UBM

EMBEDDED SYSTEMS CONFERENCE '16

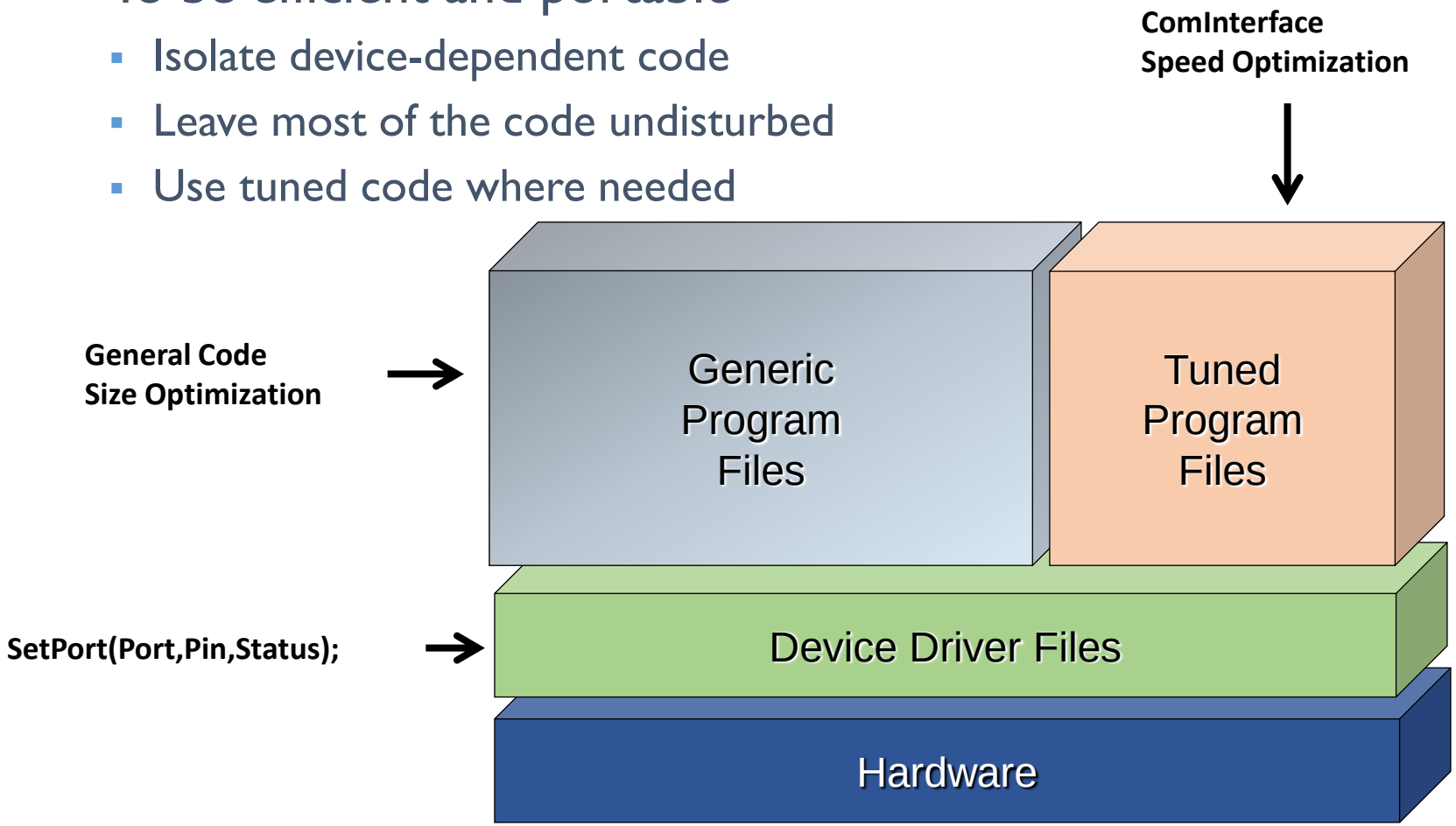
#ESCsv

Agenda

- How to structure your code
- Using correct data sizes
- Signed vs. Unsigned integers
- Floating point vs. Fixed point
- Structures
- Global variables
- Taking addresses
- Local variables
- Be careful with comparisons/calculations
- Varargs
- Function prototypes
- Static vs. volatile
- Clever code
- Saving stack

Structuring your application

- To be efficient and portable
  - Isolate device-dependent code
  - Leave most of the code undisturbed
  - Use tuned code where needed



Use “natural” data sizes

- Different architectures have different “natural” data sizes
  - Different available memories, sizes, etc.

- Using an “unnatural” data size might cost
  - A 32-bit MCU might need to shift, mask and sign-extend operations to use smaller types
  - A 32-bit MCU will need to store 64-bit data in multiple registers to hold all its contents or perform operations in RAM

- Use a natural size unless there is a compelling reason not to
  - Perhaps you are doing I/O and you need a precise number of bits
  - Bigger types might take up too much room.

```
int32[1024] > char8[1024]
```

Cost of using unnatural data sizes

```
char test_char(char a, char b)
{
  return(a+b);
}

// ARM Cortex-M (32 bit)
// ADDS      R0,R0,R1
// UXTB      R0,R0
```

2 Cycles

```
int test_int(int a, int b)
{
  return(a+b);
}

// ARM Cortex-M (32 bit)
// ADDS      R0,R1,R0
```

1 Cycle

A single cycle executed once may not have a deleterious effect on your application; however, if this function is called repeatedly, you will see performance degradation. Moreover, if you use these types of unnatural data sizes throughout your program, you will also see the size of your application grow unnecessarily.

Using fast types to get appropriate data sizes

- Use appropriate data size:
  - 8-bit operations less efficient on 32-bit CPU
  - 32-bit operations hard for 8-bit CPU
- Use another typedef to adjust
  - “int_fastX_t”: value fits in X bits
  - Use for loop counters, state variables, etc.
- Code is same and efficient across machines
- Are included as part of <stdint.h>

32-bit machine

```
typedef int    int_fast8_t;
typedef int    int_fast16_t;
typedef int    int_fast32_t;
```

64-bit machine

```
typedef long   int_fast8_t;
typedef long   int_fast16_t;
typedef long   int_fast32_t;
```

Signed or unsigned?

Think about signedness!

Signed

- Negative values possible
- Arithmetic operations always performed
- Operations will never be cheaper, but in many cases more expensive

+ - * / %

Unsigned

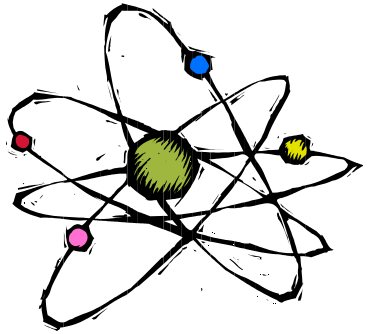
- Negative values impossible
- Bit operations are consistent
- Arithmetic operations may be optimized to bit-operations.

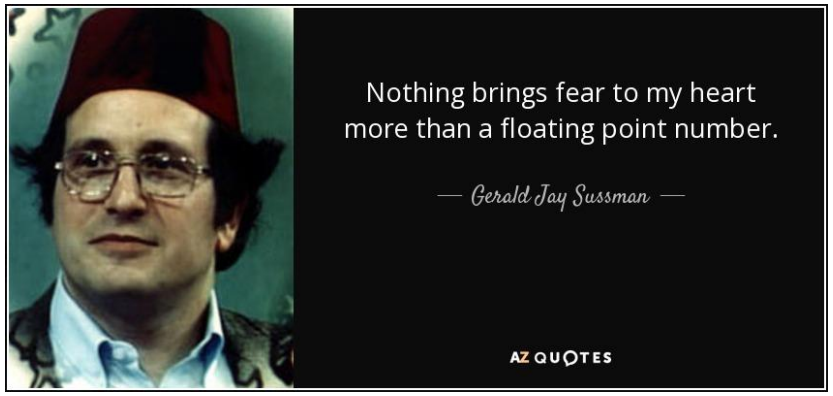
<< >> | & ^ ~

Unless you need to handle negative numbers, use unsigned types

Floating point numbers

- IEEE 754: float
  - Wide range: Float 10^{-38} to 10^{38} , Double 10^{-308} to 10^{308}
  - Good precision: Float $10^{-7,}$ Double 10^{-16}
  - Designed for giving small error in complex computations
  - Expensive in size and speed...unless there's hardware support
- "Real-world" data usually have:
  - Fixed range
  - Limited precision is available/needed
- Fixed-point arithmetic:
  - Implemented using integers
  - Can give significant savings (size *and* speed)
- Use "relaxed floating-point semantics"





Integer or floating point?

- Floating point is very expensive
  - Arithmetic is more complex
  - Brings in large library (from C runtime)
  - Could require other functionality to be more complex
    - printf( ) is ~3 times larger
    - Scanf( ) is also ~3x larger
  - Use only when really needed
- Can be done inadvertently
  - Example code:

```
#define Other 20
#define ImportantRatio          (1.05 * Other)
#define ImportantRatioBetter    ((int)(1.05 * Other))

int i=a + b * ImportantRatio;          //Will include FPLIB
int i=a + b * ImportantRatioBetter;    // Integer
```

Be careful with comparisons!

- Confusing integral promotion
  - 8-bit char, 16-bit int

```
void f1(unsigned char c1)
{
     if (c1 == ~0x80)
     {…
}
```

Test always false

What actually is done

```
void f1(unsigned char c1)
{
     if ((int)c1 == 0xFF7F)
     {…
}
```

https://www.iar.com/support/tech-notes/general/integral-types-and-possibly-confusing-behavior/

Be careful with comparisons!

- They can cancel optimizations

```
void f0(unsigned int c)
{
  unsigned int i;
  for (i = 0; i <= c; ++i)
  {
    a[i] = b[i];
  }
}
```

What if `c == UINT_MAX`?

- `i` will reach `UINT_MAX` and wrap around, thus making an infinite loop
- Optimizer must assume that this can happen, so it cancels several loop optimizations

Avoid using <= in loop tests!

Be careful with calculations!

- Confusing implicit casting

Bit shift an unsigned 32 bit object 15 times

```
uint32_t a = 0;
a = (1 << 15);
```
⟶ a = 0xFFFF8000

```
uint32_t b = 0;
b = (1 << 15u);
```
⟶ b = 0x00008000

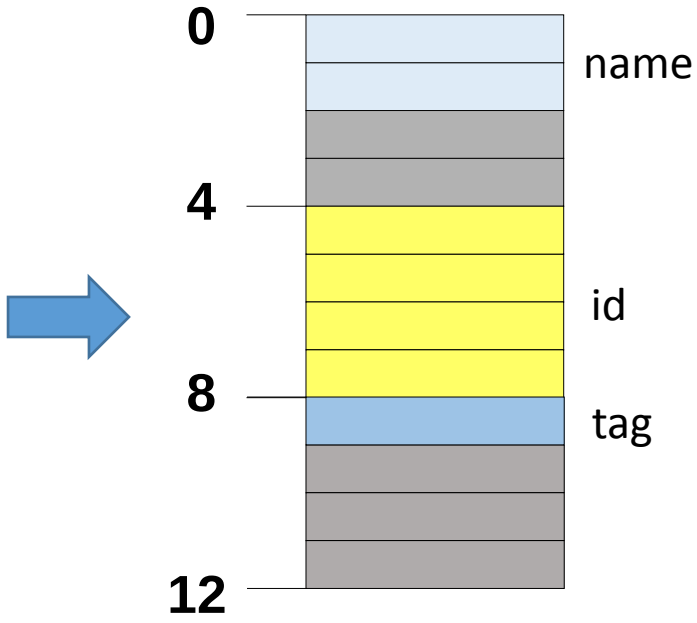
`(1 << 15)` performed as signed integer → 0x8000 = - 32768
Casted to signed long preserving value -32768 → 0xFFFF8000
Casted to unsigned long → 0xFFFF8000

The C Standard 6.3.1.1:
If an int can represent all values of the original type, the value is converted to an int; otherwise, it is converted to an unsigned int. These are called **integral promotions**.

Structures

- Structures are sometimes padded, e.g.:
  - When CPU requires alignment
  - When optimization is set for speed

```
struct record_a
{
   uint16_t name;
   uint32_t id;
   uint8_t tag;
};
```

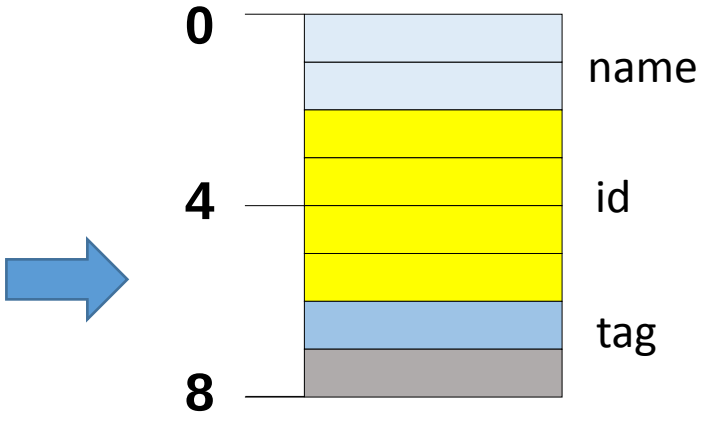


Padding to align "word"
thus, **record_a** 12 bytes

Structures

- Changing the alignment to minimize the size

```
#pragma pack(1)
struct record_b
{
   uint16_t name;
   uint32_t id;
   uint8_t tag;
};
#pragma pack()
```



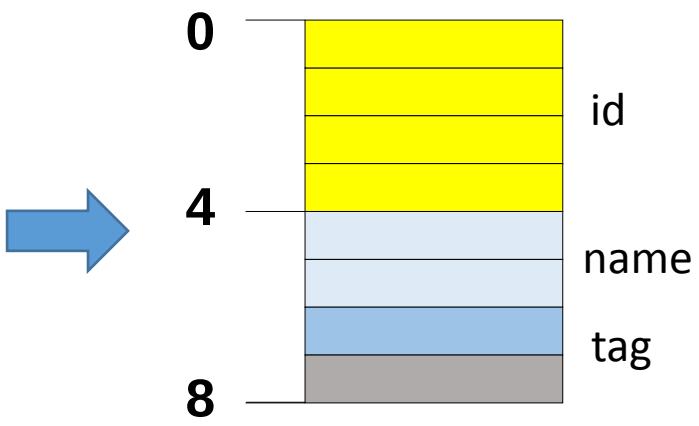
This can result in significantly larger and slower code when accessing members of the structure.

Structuring your structures

Guideline: Order fields by size

- Largest objects first
- Smallest objects last
- Automatically packs your structure efficiently!

```
struct record_c
{
   uint32_t id;
   uint16_t name;
   uint8_t tag;
};
```



Structuring your structures

```
struct record_a
{
  uint16_t name;
  uint32_t id;
  uint8_t tag;
};

struct record_a A;

void setA()
{
  A.name = 'A';
  A.id = 1;
  A.tag = 0x01;
}
```

A = 12 byte
setA() = 20 byte
11 cycles
} 32

```
#pragma pack(1)
struct record_b
{
  uint16_t name;
  uint32_t id;
  uint8_t tag;
};
#pragma pack()

struct record_b B;

void setB()
{
  B.name = 'B';
  B.id = 2;
  B.tag = 0x02;
}
```

B = 8 byte
setB() = 24 byte
22 cycles
} 32

```
struct record_c
{
  uint32_t id;
  uint16_t name;
  uint8_t tag;
};
…
struct record_c C;

void setC()
{
  C.name = 'C';
  C.id = 3;
  C.tag = 0x03;
}
```

C = 8 byte
setC() = 20 byte
11 cycles
} 28

Using global variables

Advantages:

- Accessible from the whole program
- Easy to understand and use

Drawbacks:

- Will not be placed in a register, operations will be slow
- Any function call may change the value
- Compiler can’t be as aggressive

Solution = Copy the value into a local variable

- Local variable will probably be placed in register, so operations on it is fast
- Function calls will not change the value
- Write back the value if needed.

Example of using temp variable with global

Before:

```
uint8_t gGlobal;

void foo(int32_t x)
{
   ...
   bar(gGlobal);    // bar may change gGlobal
   ...
   gGlobal++;                  // Memory read and write
operation
   ...
}
```

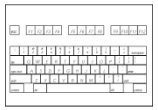
After:

```
uint8_t gGlobal;

void foo(int32_t x)
{
   uint8_t cTemp=gGlobal;     // cTemp probably placed in register
   ...
   bar(cTemp);                                    // bar cannot change
gGlobal
   ...
   cTemp++;                                       // Fast update of
register
   ...
   gGlobal=cTemp;             // Store the result back
}
```

Taking addresses

- Address taken → uses memory
- Cannot be register-allocated

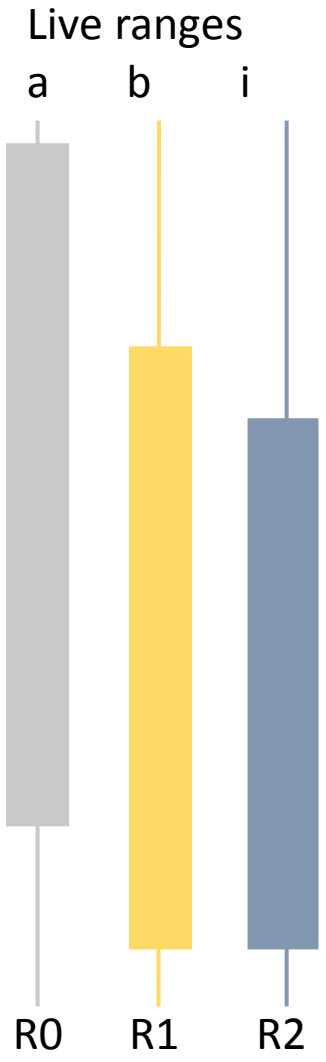
```
int a;
...
scanf("%d",&a);

Func1(a);
…
FuncN(a);
```



```
int a, temp;
...
scanf("%d",&temp);
a=temp;
/*After this temp is dead, and the local
register-placed 'a' is used*/

Func1(a);
…
FuncN(a);
```

Registers and locals

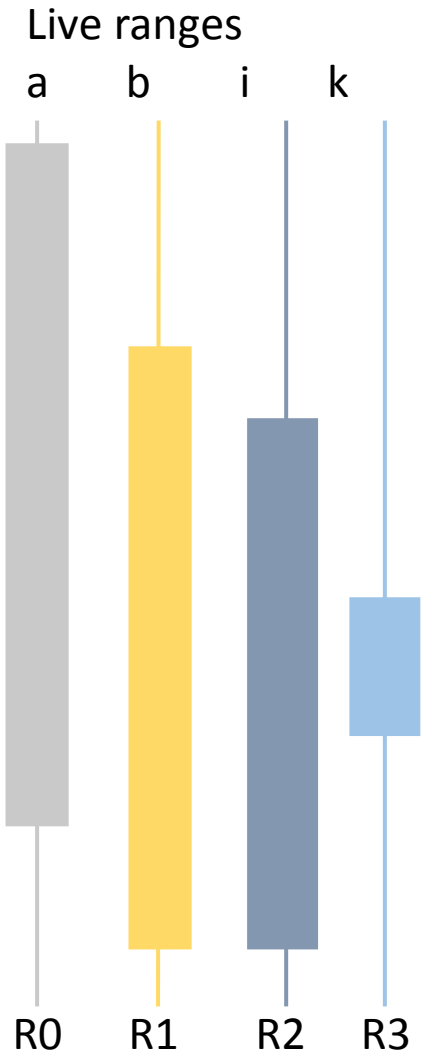
```
void foo(char a)
{
  char b,i;

  b=a<<3;
  for(i=0;… )
    OUTPUT(b);




  b=a<<5;
  for(i=0;… )
    OUTPUT(b);
}
```



- Register usage
  - Variables can share a register
  - Only present in register while “live”

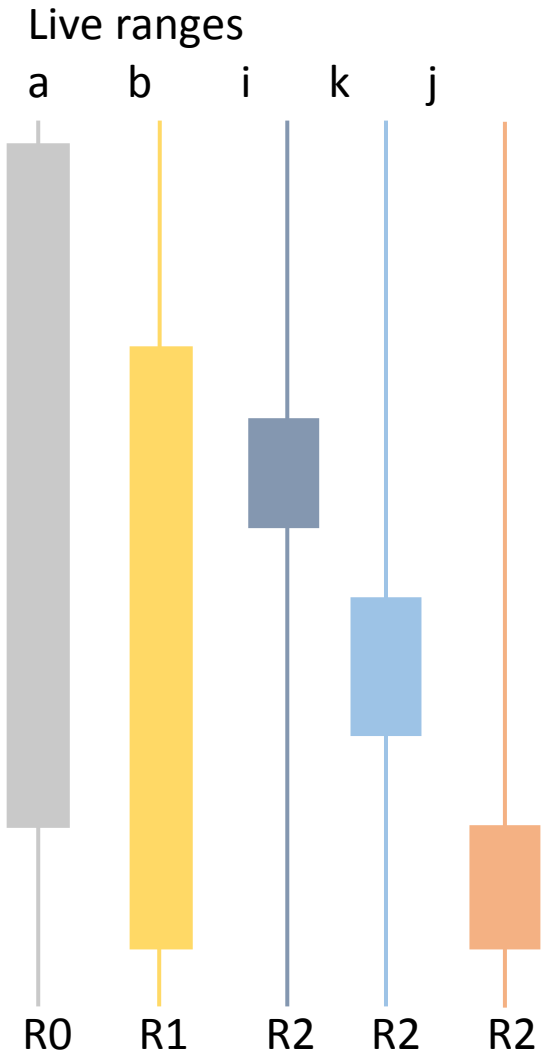
Registers and locals

```
void foo(char a)
{
  char b,i,k;

  b=a<<3;
  for(i=0;… )
    OUTPUT(b);

  k=a<<4
    OUTPUT(k);

  b=a<<5;
  for(i=0;… )
    OUTPUT(b);
}
```



Registers and locals

```
void foo(char a)
{
  char b,i,k,j;

  b=a<<3;
  for(i=0;… )
    OUTPUT(b);

  k=a<<4
    OUTPUT(k);

  b=a<<5;
  for(j=0;… )
    OUTPUT(b);
}
```



- Don’t worry about “extra” variables

Varargs

- Variable number of arguments
  - `int printf(char *, ...)`
  - Special macros to access arguments
- Forces arguments to stack
  - Step through them using pointers
- No parameters in registers
- Source code gets more complex
- Error-prone
- Object code gets bigger
- Bad idea, simply

Function prototypes

- C and function calls:
  - No declaration: call out of the blue
  - K&R declarations: `void foo();`
  - ANSI prototypes: `char bar(short);`
- Without ANSI prototypes:
  - No proper type-checking
  - All arguments promoted: casting code and parameter data is larger = more stack used
- Always use ANSI prototypes!
  - Turn on checking in compiler!
- ELF has no type checking at link time
  - You can make sure you use the same prototype everywhere by using a header file.

Static vs. volatile

- The keyword `static` means that all references to a variable are known
  - It also means that the compiler/linker will assign a RAM address for the variable
  - The variable can only be used in the source file where it was declared

- The keyword `volatile` means that all accesses to a variable must be preserved
  - Useful when you're doing shared access of a variable
  - Also handy for trigger access, where accessing the variable is a stimulus for doing some other action
  - Can be used for modified access, i.e. where the contents of the variable can change in ways not known to the compiler

Static vs. volatile

- Is it atomic?

Never assume that volatile means atomic.

```
volatile int32_t vol = 1;
void f5()
{
  vol++; /* Is this atomic? */
}

LDR.N    R0,??DataTable7 ;;vol address
LDR      R1,[R0, #+0]    ;; Atomic
ADDS     R1,R1,#+1       ;; ADDS…
STR      R1,[R0, #+0]    ;; Atomic
BX       LR              ;; return
```

Static vs. volatile

- The “empty loop” pitfall
  - Empty delay loop does not work
    - Code with no effect
    - Gets removed
    - Delay is always zero
  - To really delay:
    - Access volatile variable
    - Use OS services
    - Use CPU timer

```
void delay(int time)
{
  int i;
  for (i=0;i<time;i++)
  {}
  return;
}

void SetHW(void)
{
  SET_DIRECTION(0x20);
  delay(100);
  OUT_SIGNAL(0xB4);
  delay(200);
  READ_PORT(0x02)
}
```

Don’t write “clever” code

- Straightforward code
  - Easy to read → Easy to maintain
  - Better optimization & better tested

Clever solution

```
unsigned long int a;
unsigned char b;
b|=!!(a<<11);
```

```
a = (b == 3) ? 4 : 5;
```

Better solution

```
unsigned long int a;
unsigned char b;
if((a & 0x1FFFFF) != 0)
  b |= 0x01;
```

```
if (b == 3)
  a = 4;
else
  a = 5;
```

Comments on “clever” code

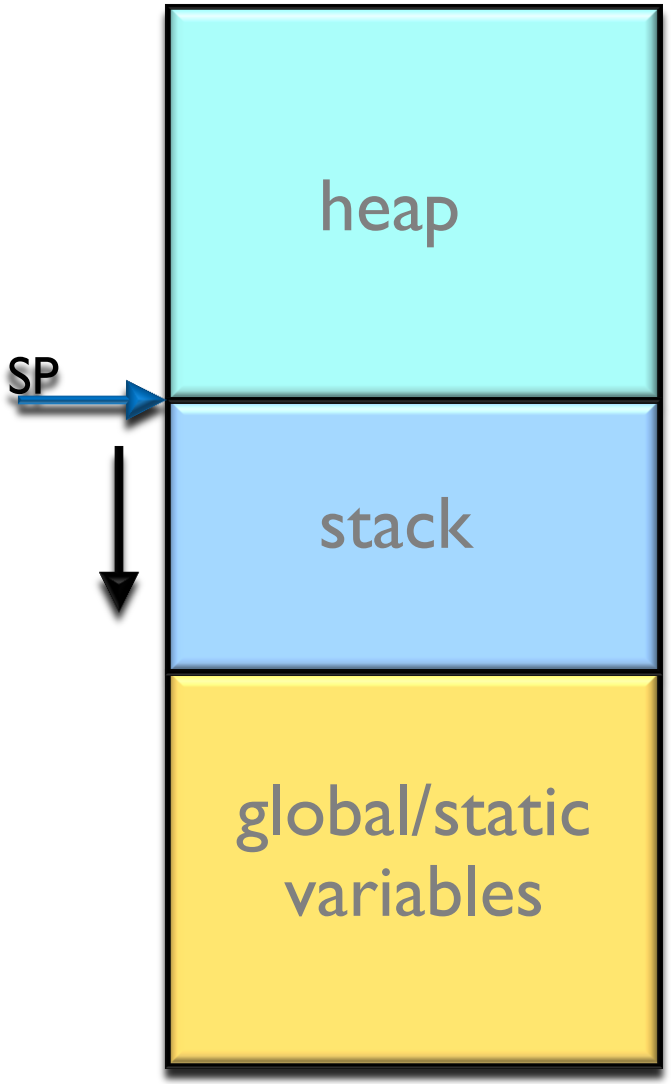
“If understanding the code requires knowledge of page 543 of the spec, then chances are nobody will understand the code. I prefer simple and understandable.”

“The more you stray from the well worn paths, the more likely you'll write code the compiler misunderstands and either gets wrong or optimizes poorly.”

“What has precedence between || and &?
Correct answer is: who cares! Don't write code like that! “

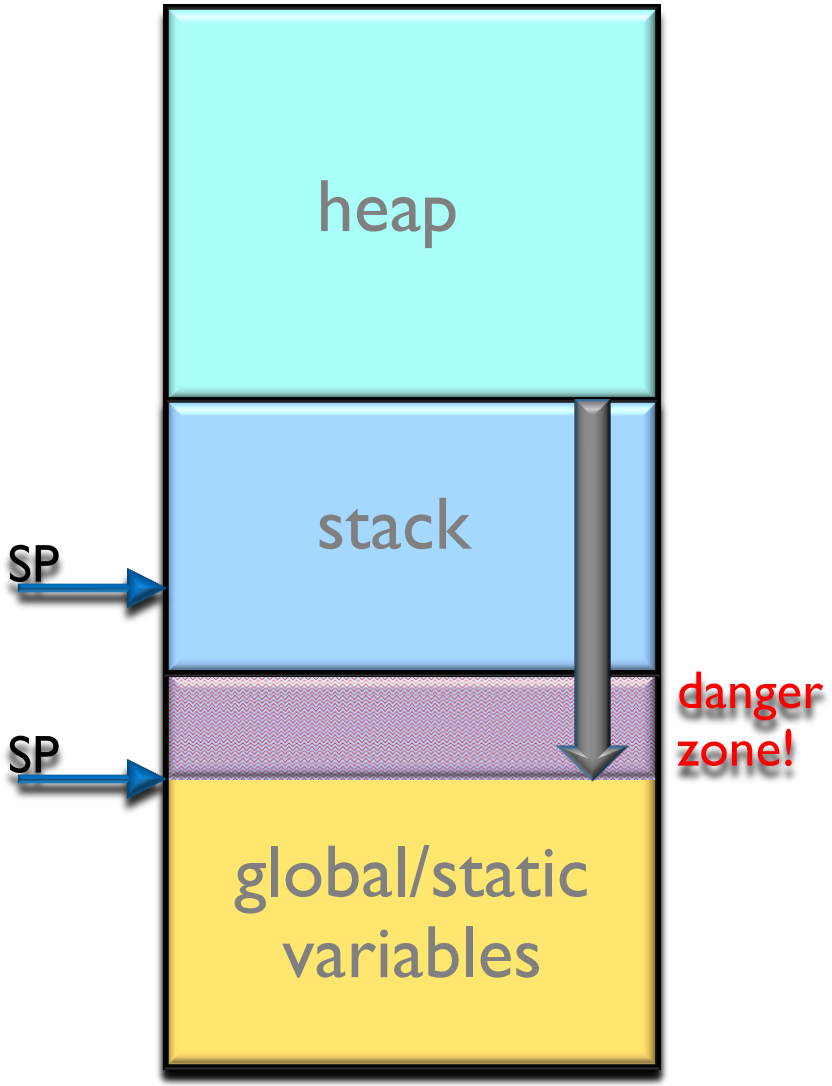
The stack

- The stack is used for:
  - Local variables
  - Return addresses
  - Function arguments
  - Compiler temporaries
  - Interrupt contexts
- Life span is the duration of the function



Stack overflow

- Stack overflow
  - there is no protection on SP
  - the stack grows into the global area overwriting application data
    - corrupted variables
    - wild pointers
    - corrupted return addresses
- Errors are really hard to catch!
- Setting the stack size
  - too small - overflow
  - too big - waste of memory



Avoiding stack overflow

- Test and measurement methods
  - Track the stack pointer
  - Use stack guard zones
  - Fill the stack area with an arbitrary bit pattern

- Calculation methods
  - Manual calculation
  - Static stack calculation tool

Avoiding stack overflow

- Avoid `printf()` and its relatives…
- Pass by reference instead of by copy
  - At least for objects of size greater than the register size
- Limit the number of arguments to a function
  - The ABI of an MCU tells a compiler how many arguments can be passed in the registers, all others must be passed on the stack
  - *void doThisOrThat(…,doWhat);* →
    void doThis(…);
    void doThat(…);

Summary

Thank You!

Questions?

UBM

EMBEDDED SYSTEMS CONFERENCE '16

@ESC_Conf
#ESCsv