```
volatile unsigned short DMA1SA @ 0x01eau;

void iar_buggy_func(unsigned char ch) {
    DMA1SA = (uint16_t)&ch;
}
```

- Compiled to:

```
iar_buggy_func:
    decd.w SP
    mov.w SP,&DMA1SA
    incd.w SP
    ret
```

- Response from compiler support:

  To condense the problem, we have a function that looks like the following:

```
volatile unsigned char *v;
void iar_buggy_func (unsigned char ch) {
  v = & ch;
}
```

- Ok so far?

- More from compiler support:

  In the report you correctly noted that the value of “ch” was never written to the stack. When it comes to "volatile", the basic rule is that anything that has some kind of side-effect or could be accessed by the underlying hardware should be declared to be “volatile”. In this case, when writing to "v", both "v" and "ch" is accessed by the hardware. Hence, both should be declared "volatile".

- Is this correct?

- More from compiler support:

  As "ch" is a parameter, it is possible to assign it to a local volatile variable before the assignment, for example:

```
volatile unsigned char *v;
void iar_buggy_func (unsigned char ch)
{
  volatile unsigned char ch2 = ch;
  v = & ch2;
}
```

- Is this a good fix?

Last Time

- Language subsets
  - Avoid problematic constructs
  - Improve maintainability
- MISRA-C
  - C subset for critical software
  - Much of MISRA is a good idea anyway

Today

- How to deal with limited RAM
- C extensions for embedded systems
  - Things that need to be added to C to make it a better embedded language

Cost of Nearly Full Resources

- Koopman p. 178:
  - Software costs rise dramatically when system resources are more than 75% to 85% full.
- When resources approach 95%, development becomes extremely difficult
- Tradeoff:
  - Buying a bigger part reduces NRE costs
  - But increases unit cost

Rules of Thumb

- Get bigger hardware if
  - Production run is less than 1 million units
  - Resources are >80% full
- If production run is less than 10,000 units, oversize resources by a factor of 2
- Today: RAM
  - Later: CPU time and network bandwidth
  - Of course there are other resources that matter

Memory Limits

- PC programs that use too much memory run slowly
  - Virtual memory system provides "soft failure"
    - More and more paging, until finally somebody kills the program
- In embedded systems:
  - Hard failures – crashes – are the more likely result of memory exhaustion
    - Often no VM subsystem
    - Less RAM in the first place
  - Even when there is VM, soft failure can be a real problem
    - Nobody around to kill the thrashing process
    - Running slowly unacceptable in a real-time system

How is RAM allocated?

- Statically
- Dynamically on the stack
- Dynamically on the heap
- Question 1: What is the total worst-case RAM requirement of your system?
- Question 2: Is this larger than the amount of RAM available?

- Reality:
  - Most programmers don't check malloc() return value for non-null
  - Not just laziness – often there's just no way to handle this
- In small embedded systems – like ours – probably a bad idea to have a heap at all
  - Heaps introduce many failure points into systems that should not fail
  - Failure points make it hard to reason about software

When is a heap allowed?

1. When you can be absolutely sure that allocations will succeed
   - Requires computing the maximum heap utilization of a program
   - Obviously there must be no memory leaks!
   - Fragmentation makes the computation even more difficult
2. When allocation failure can be handled gracefully
   - Almost never
3. (Maybe) when allocation is only done at boot time

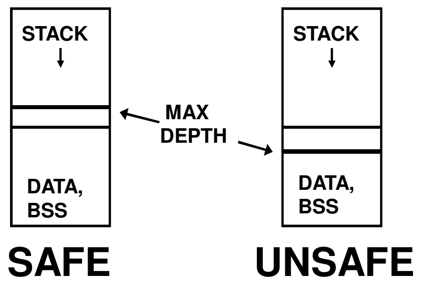
Heap Alternatives

- Static allocation
  - I.e., global variables
  - Efficient in terms of cycles
  - Can be wasteful in terms of bytes
- Overlays
  - I.e., manual reuse of memory regions
  - Can be very efficient in terms of cycles and bytes
  - But very difficult to get right
  - Apollo example
- Stack allocation
  - Efficient in terms of cycles and bytes
  - Memory usage patterns often don't match stack semantics
  - Stacks can overflow too

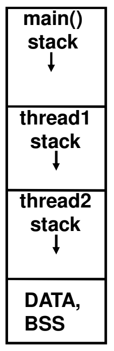
Stack Overflow

- Stack must contain enough RAM to not overflow in the worst case
- Complications
  - When there are multiple stacks, none of them must overflow
    - Threads have their own stacks
    - ARM has multiple hardware stacks
  - Stack depth usually depends on interrupt behavior
    - Hence overflows are unpredictable
    - Robot example
  - Recursion is hard to think about
    - Especially beware unintentional recursion

Stack Overflow w/o Threads



Stack Overflow with Threads



•All stacks in the system must be big enough

•Question: From which stack do interrupt handlers get their stack memory?

Interrupts and the Stack

- Interrupt handlers use stack memory
- For nested interrupts, you total up the stack requirements for all handlers
- For non-nested interrupts, you take the maximum stack requirement of any handler

Avoiding Stack Overflow

- Ways to estimate maximum stack extent:
  - Testing
  - Static analysis
- Always true
  - Worst depth seen in testing ≤ true worst-case depth ≤ depth predicted by static analysis
- Goal: Stack just large enough
  - Too large: wasted RAM
  - Too small: occasional memory corruption
  - Way too small: can't even boot the system

Stack Depth Testing

1. **Insert explicit checks on stack depth into your code**
   - How reliable is this method?
2. **Check the stack pointer in a periodic interrupt handler**
   - How reliable is this method?
3. **"Red zone" technique**
   - Initialize all stack memory to known values
   - Check how many of these get overwritten
   - How reliable is this method?

Analyzing Stack Depth

- **Options:**
  - Stack analysis tool
  - Stack analysis by hand
- **Stack analysis by hand:**
  - Trace through each function, looking for functions that affect stack depth
    - You need to find the worst case stack depth for this function
  - Trace through the call graph for your application, adding up the stack depths for each function
    - You need to find the worst-case stack depth for the entire application

Link and Unlink

- **Link instruction:**
  - Pushes the contents of the specified address register onto the stack
  - Loads the updated stack pointer into the address register
  - Adds displacement to stack pointer
- **Unlink instruction:**
  - Load stack pointer from specified address register
  - Load the address register with the longword pulled from top of stack

Stack Analysis

```
init_porttc:
0x00000000  link     a6,#0
0x00000004  moveq    #0,d0
0x00000006  move.b   d0,___IPSBAR+1048687
0x0000000C  moveq    #15,d0
0x0000000E  move.b   d0,___IPSBAR+1048615
0x00000014  moveq    #0,d0
0x00000016  move.b   d0,___IPSBAR+1048591
0x0000001C  unlk     a6
0x0000001E  rts
```

Stack Analysis

```
init_porttc:                 ←——————————— 4
0x00000000  link     a6,#0
0x00000004  moveq    #0,d0
0x00000006  move.b   d0,___IPSBAR+1048687
0x0000000C  moveq    #15,d0
0x0000000E  move.b   d0,___IPSBAR+1048615
0x00000014  moveq    #0,d0
0x00000016  move.b   d0,___IPSBAR+1048591
0x0000001C  unlk     a6
0x0000001E  rts
```

Stack Analysis

```
init_porttc:                 ←——————————— 4
0x00000000  link     a6,#0 ←——————————————— 8
0x00000004  moveq    #0,d0
0x00000006  move.b   d0,___IPSBAR+1048687
0x0000000C  moveq    #15,d0
0x0000000E  move.b   d0,___IPSBAR+1048615
0x00000014  moveq    #0,d0
0x00000016  move.b   d0,___IPSBAR+1048591
0x0000001C  unlk     a6
0x0000001E  rts
```

Stack Analysis

```
init_porttc:                    <-------------------- 4
0x00000000  link      a6,#0
                                   <-------------------- 8
0x00000004  moveq     #0,d0
0x00000006  move.b    d0,____IPSBAR+1048687
0x0000000C  moveq     #15,d0
0x0000000E  move.b    d0,____IPSBAR+1048615
0x00000014  moveq     #0,d0
0x00000016  move.b    d0,____IPSBAR+1048591
                                   <-------------------- 8
0x0000001C  unlk      a6
                                <-------------------- 4
0x0000001E  rts
                                   <-------------------- 0
```

Stack Analysis

```
init_porttc:                    <-------------------- 4
0x00000000  link      a6,#0
                                   <-------------------- 8
0x00000004  moveq     #0,d0
0x00000006  move.b    d0,____IPSBAR+1048687
0x0000000C  moveq     #15,d0
0x0000000E  move.b    d0,____IPSBAR+1048615
0x00000014  moveq     #0,d0
0x00000016  move.b    d0,____IPSBAR+1048591
                                   <-------------------- 8
0x0000001C  unlk      a6
                                <-------------------- 4
0x0000001E  rts
                                   <-------------------- 0
```

Invariant: Function always has zero net effect on the stack

```
 _handler_irq4:
0x00000000  link     a6,#0
0x00000004  lea      -28(a7),a7
0x00000008  movem.l  d0-d2/a0-a1,8(a7)
0x0000000E  lea      ___IPSBAR+1048591,a0
0x00000014  move.l   a0,-24(a6)
0x00000018  movea.l  -24(a6),a1
0x0000001C  movea.l  -24(a6),a0
0x00000020  move.b   (a0),d1
0x00000022  moveq    #0,d0
0x00000024  move.b   d1,d0
0x00000026  eori.l   #0x2,d0
0x0000002C  move.b   d0,(a1)
0x0000002E  lea      _@78,a0
0x00000034  move.l   a0,(a7)
0x00000036  jsr      _printf
0x0000003C  lea      ___files+70,a0
0x00000042  move.l   a0,(a7)
0x00000044  jsr      _fflush
0x0000004A  moveq    #16,d0
0x0000004C  move.b   d0,___IPSBAR+1245190
0x00000052  movem.l  8(a7),d0-d2/a0-a1
0x00000058  unlk     a6
0x0000005A  rte
```

```
 _handler_irq4:                <-------------------- 0+8 = 8
0x00000000  link     a6,#0
0x00000004  lea      -28(a7),a7
0x00000008  movem.l  d0-d2/a0-a1,8(a7)
0x0000000E  lea      ___IPSBAR+1048591,a0
0x00000014  move.l   a0,-24(a6)
0x00000018  movea.l  -24(a6),a1
0x0000001C  movea.l  -24(a6),a0
0x00000020  move.b   (a0),d1
0x00000022  moveq    #0,d0
0x00000024  move.b   d1,d0
0x00000026  eori.l   #0x2,d0
0x0000002C  move.b   d0,(a1)
0x0000002E  lea      _@78,a0
0x00000034  move.l   a0,(a7)
0x00000036  jsr      _printf
0x0000003C  lea      ___files+70,a0
0x00000042  move.l   a0,(a7)
0x00000044  jsr      _fflush
0x0000004A  moveq    #16,d0
0x0000004C  move.b   d0,___IPSBAR+1245190
0x00000052  movem.l  8(a7),d0-d2/a0-a1
0x00000058  unlk     a6
0x0000005A  rte
```

```
 _handler_irq4:                <-------------------- 0+8 = 8
0x00000000  link     a6,#0        <-------------------- 8+4 = 12
0x00000004  lea      -28(a7),a7
0x00000008  movem.l  d0-d2/a0-a1,8(a7)
0x0000000E  lea      ___IPSBAR+1048591,a0
0x00000014  move.l   a0,-24(a6)
0x00000018  movea.l  -24(a6),a1
0x0000001C  movea.l  -24(a6),a0
0x00000020  move.b   (a0),d1
0x00000022  moveq    #0,d0
0x00000024  move.b   d1,d0
0x00000026  eori.l   #0x2,d0
0x0000002C  move.b   d0,(a1)
0x0000002E  lea      _@78,a0
0x00000034  move.l   a0,(a7)
0x00000036  jsr      _printf
0x0000003C  lea      ___files+70,a0
0x00000042  move.l   a0,(a7)
0x00000044  jsr      _fflush
0x0000004A  moveq    #16,d0
0x0000004C  move.b   d0,___IPSBAR+1245190
0x00000052  movem.l  8(a7),d0-d2/a0-a1
0x00000058  unlk     a6
0x0000005A  rte
```

```
 _handler_irq4:                <-------------------- 0+8 = 8
0x00000000  link     a6,#0        <-------------------- 8+4 = 12
0x00000004  lea      -28(a7),a7        <------------ 12+28 = 40
0x00000008  movem.l  d0-d2/a0-a1,8(a7)
0x0000000E  lea      ___IPSBAR+1048591,a0
0x00000014  move.l   a0,-24(a6)
0x00000018  movea.l  -24(a6),a1
0x0000001C  movea.l  -24(a6),a0
0x00000020  move.b   (a0),d1
0x00000022  moveq    #0,d0
0x00000024  move.b   d1,d0
0x00000026  eori.l   #0x2,d0
0x0000002C  move.b   d0,(a1)
0x0000002E  lea      _@78,a0
0x00000034  move.l   a0,(a7)
0x00000036  jsr      _printf
0x0000003C  lea      ___files+70,a0
0x00000042  move.l   a0,(a7)
0x00000044  jsr      _fflush
0x0000004A  moveq    #16,d0
0x0000004C  move.b   d0,___IPSBAR+1245190
0x00000052  movem.l  8(a7),d0-d2/a0-a1
0x00000058  unlk     a6
0x0000005A  rte
```

```
 _handler_irq4:                          ← 0+8 = 8
0x00000000  link     a6,#0              ← 8+4 = 12
0x00000004  lea      -28(a7),a7         ← 12+28 = 40
0x00000008  movem.l  d0-d2/a0-a1,8(a7)  ← 40 + 0 = 40
0x0000000E  lea      ___IPSBAR+1048591,a0
0x00000014  move.l   a0,-24(a6)
0x00000018  movea.l  -24(a6),a1
0x0000001C  movea.l  -24(a6),a0
0x00000020  move.b   (a0),d1
0x00000022  moveq    #0,d0
0x00000024  move.b   d1,d0
0x00000026  eori.l   #0x2,d0
0x0000002C  move.b   d0,(a1)
0x0000002E  lea      _@78,a0
0x00000034  move.l   a0,(a7)
0x00000036  jsr      _printf
0x0000003C  lea      ___files+70,a0
0x00000042  move.l   a0,(a7)
0x00000044  jsr      _fflush
0x0000004A  moveq    #16,d0
0x0000004C  move.b   d0,___IPSBAR+1245190
0x00000052  movem.l  8(a7),d0-d2/a0-a1
0x00000058  unlk     a6
0x0000005A  rte
```

```
 _handler_irq4:                          ← 0+8 = 8
0x00000000  link     a6,#0              ← 8+4 = 12
0x00000004  lea      -28(a7),a7         ← 12+28 = 40
0x00000008  movem.l  d0-d2/a0-a1,8(a7)  ← 40 + 0 = 40
0x0000000E  lea      ___IPSBAR+1048591,a0
0x00000014  move.l   a0,-24(a6)
0x00000018  movea.l  -24(a6),a1
0x0000001C  movea.l  -24(a6),a0
0x00000020  move.b   (a0),d1
0x00000022  moveq    #0,d0
0x00000024  move.b   d1,d0
0x00000026  eori.l   #0x2,d0
0x0000002C  move.b   d0,(a1)
0x0000002E  lea      _@78,a0
0x00000034  move.l   a0,(a7)
0x00000036  jsr      _printf
0x0000003C  lea      ___files+70,a0
0x00000042  move.l   a0,(a7)
0x00000044  jsr      _fflush
0x0000004A  moveq    #16,d0
0x0000004C  move.b   d0,___IPSBAR+1245190
0x00000052  movem.l  8(a7),d0-d2/a0-a1
0x00000058  unlk     a6
0x0000005A  rte
```

```
 _handler_irq4:                          ← 0+8 = 8
0x00000000  link     a6,#0              ← 8+4 = 12
0x00000004  lea      -28(a7),a7         ← 12+28 = 40
0x00000008  movem.l  d0-d2/a0-a1,8(a7)  ← 40 + 0 = 40
0x0000000E  lea      ___IPSBAR+1048591,a0
0x00000014  move.l   a0,-24(a6)
0x00000018  movea.l  -24(a6),a1
0x0000001C  movea.l  -24(a6),a0
0x00000020  move.b   (a0),d1
0x00000022  moveq    #0,d0
0x00000024  move.b   d1,d0
0x00000026  eori.l   #0x2,d0
0x0000002C  move.b   d0,(a1)
0x0000002E  lea      _@78,a0
0x00000034  move.l   a0,(a7)
0x00000036  jsr      _printf
0x0000003C  lea      ___files+70,a0
0x00000042  move.l   a0,(a7)
0x00000044  jsr      _fflush
0x0000004A  moveq    #16,d0
0x0000004C  move.b   d0,___IPSBAR+1245190
0x00000052  movem.l  8(a7),d0-d2/a0-a1
0x00000058  unlk     a6                 ← 8
0x0000005A  rte
```

```
 _handler_irq4:                          ← 0+8 = 8
0x00000000  link     a6,#0              ← 8+4 = 12
0x00000004  lea      -28(a7),a7         ← 12+28 = 40
0x00000008  movem.l  d0-d2/a0-a1,8(a7)  ← 40 + 0 = 40
0x0000000E  lea      ___IPSBAR+1048591,a0
0x00000014  move.l   a0,-24(a6)
0x00000018  movea.l  -24(a6),a1
0x0000001C  movea.l  -24(a6),a0
0x00000020  move.b   (a0),d1
0x00000022  moveq    #0,d0
0x00000024  move.b   d1,d0
0x00000026  eori.l   #0x2,d0
0x0000002C  move.b   d0,(a1)
0x0000002E  lea      _@78,a0
0x00000034  move.l   a0,(a7)
0x00000036  jsr      _printf
0x0000003C  lea      ___files+70,a0
0x00000042  move.l   a0,(a7)
0x00000044  jsr      _fflush
0x0000004A  moveq    #16,d0
0x0000004C  move.b   d0,___IPSBAR+1245190
0x00000052  movem.l  8(a7),d0-d2/a0-a1
0x00000058  unlk     a6                 ← 8
0x0000005A  rte                         ← 0
```

Whole-System Stack Usage

- Compute the callgraph
  - One for main()
  - One for each interrupt
- Compute the maximum stack usage for each callgraph
- Add these numbers together if interrupts can preempt each other
- Add main() + max(all interrupts) if interrupts cannot preempt

Language Extensions

- Idea: Add support for specific embedded programming idioms to C
- Why?
  - Documentation – when reading the code you see “interrupt” instead of a bunch of assembly code
  - Correctness – once the compiler writers get it right, everyone can reuse the solution
  - Efficiency – compiler can take advantage of special knowledge to make fast code
  - Rapid development – using a language feature is easy, fighting the language is hard

Extensions in Keil 8051 C

- Locate struct at an absolute address:
  - `struct link list idata _at_ 0x40;`
- Respect a different compiler's calling convention:
  - `extern alien char plm_func (int, char);`
- Conveniently deal with a single bit:
  - `sbit mybit15 = ibase ^ 15;`
- Interrupts and register banks:
  - `void timer0 (void) interrupt 1 using 2 { … }`
- Interface with an RTOS:
  - `void func (void) _task_ num _priority_ pri { … }`

Embedded C

- Specification created by ISO – International Organization for Standardization
- Idea – create nice, portable extensions so each individual compiler vendor doesn't go off and do something different
- Features
  - Fixed point arithmetic
  - Saturating arithmetic
  - Segmented memory spaces
  - Hardware I/O addressing

Fixed Point

- Floating point arithmetic is general purpose
  - Supports a very wide range of numbers
- FP is overkill for some problems where the required range of precision is narrower
- Also: Common for embedded processors to lack a floating point unit
  - Even our MCF52233 – pretty powerful for a microcontroller – does not have one
  - FP emulation in SW is slow
- But integers cannot represent fractional values
- Solution: Fixed point arithmetic

Fixed Point

- 8 bits can represent 256 values
- However, we can interpret those 256 values in many different ways
  - Integer (or 8.0 fixed point): 0, 1, .. , 255
  - 6.2 fixed point: 0, 0.25, 0.5, .. , 63.75
  - 4.4 fixed point: 0, 0.0625, 0.125, .. , 15.9375
  - 0.8 fixed point: 0, 0.00390625, .. , 0.99609375
  - All these formats have signed equivalents
- How would you implement fixed point:
  - Comparison?
  - Addition and subtraction?
  - Multiplication and division?

Embedded C Fixed Point

- New primitive type `fract`
- Signed `fract` represents [-1.0 .. 1.0)
- Unsigned `fract` represents [0.0 .. 1.0)
- Three types of `fract`: short, default, and long
  - Moving up in types, precision is equal or better
  - Precision not defined in order to permit efficient implementations on diverse hardware

More Fixed Point

- New primitive type `accum`
  - These encode fixed-point numbers outside the range [-1..1)
- For `fract` and `accum`: Arithmetic operations are defined, bitwise operations are not
- Expressions can mix `fract`, `accum`, and regular integers

Address Spaces

- Embedded systems commonly contain different types of RAM
  - E.g. Flash, SRAM, DRAM, EEPROM, etc.
- Embedded C lets you say:
  - X int Z;
  - Allocates Z in address space X;
  - What is the alternative to address spaces?
- Pointers qualified by an address space
- Unqualified pointers must be able to point to any address space
  - Inefficient!
- ColdFire HW does not need address spaces

I/O Addressing

- Embedded C defines portable API for manipulation hardware registers:

```
unsigned int iord( ioreg_designator );
void iowr( ioreg_designator, unsigned int value );
void ioor( ioreg_designator, unsigned int value );
void ioand( ioreg_designator, unsigned int value );
void ioxor( ioreg_designator, unsigned int value );
```

- Avoids one use of volatile variables

Summary

- Dealing with finite memory is hard
  - Requires careful system design
  - Requires careful analysis and testing
  - Don't make resources nearly full!
- Language extensions are good
  - … as long as they are widely supported
  - ISO standard Embedded C helps