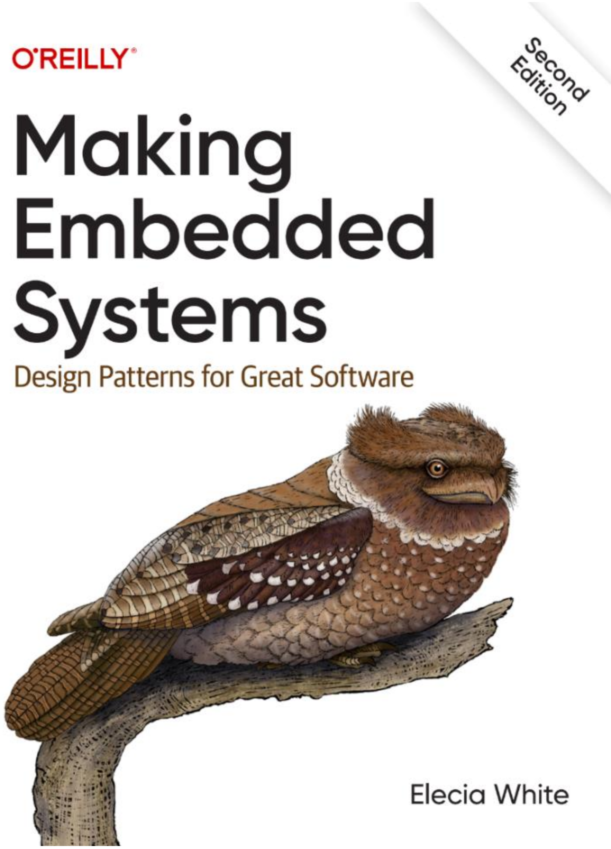
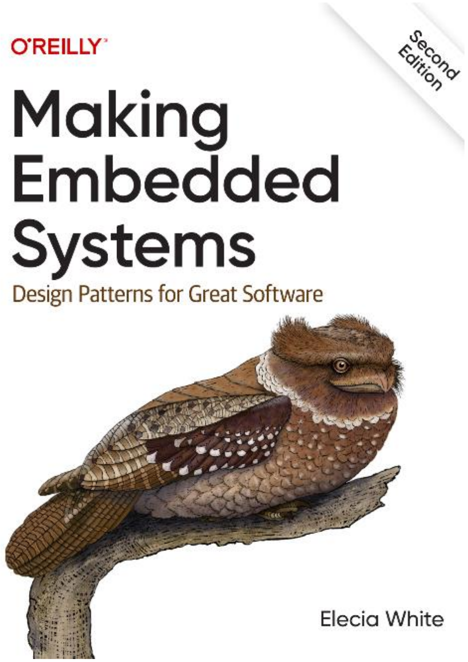


Introduction to Embedded Systems



Logical Elegance, Inc

About Me

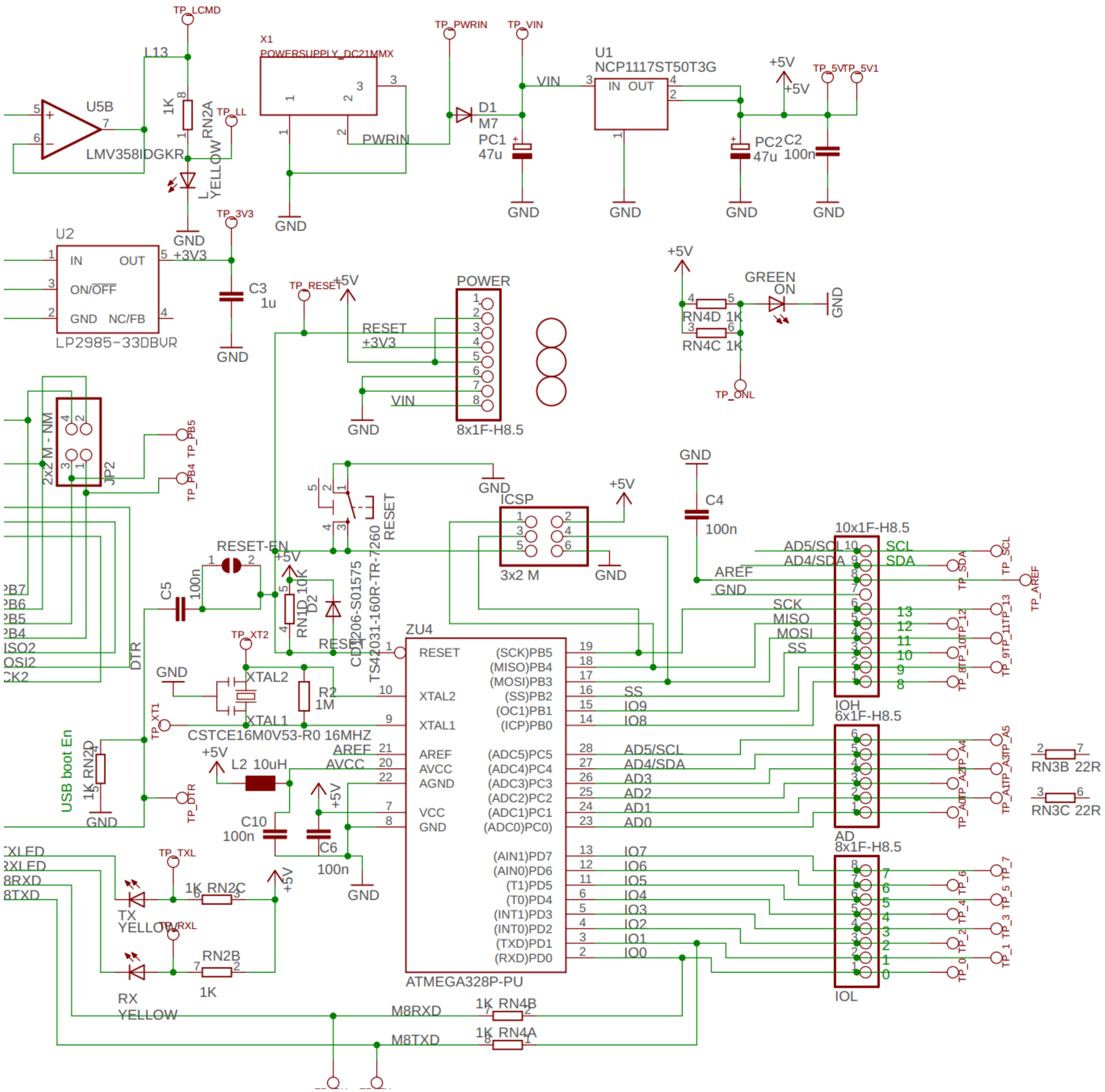
What Is an Embedded System?








Embedded Software: CS or EE?



Touching Hardware

Talking to the Processor

8.5.6 GPIO port output data register (GPIOx_ODR) (x = A to I)

Address offset: 0x14

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
OD15	OD14	OD13	OD12	OD11	OD10	OD9	OD8	OD7	OD6	OD5	OD4	OD3	OD2	OD1	OD0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **OD[15:0]:** Port output data I/O pin y (y = 15 to 0)
These bits can be read and written by software.

If you want to set a bit in a register, OR it with the register:

```
register = register | bit;
register = register | (1 << 3); // turn on the 3rd bit in the register
register |= 0x08;               // same, different syntax
```

```
*((uint32_t*) 0x80000014) |= 1 << 2; // set pin A2 high
```

STM32F103 processor

```
GPIOA->ODR |= (1 << 2);     // IOA_2 high
```

MSP430 processor

```
P1OUT |= BIT2;              // IO1_2 high
```

ATtiny processor

```
PORTB |= 0x4;               // IOB_2 high
```

Setting Registers

Bitwise operation	Meaning	Syntax	Examples

AND	If both of the two inputs have a bit set, the output will as well.	&	`0x01 & 0x02 = 0x00`
			`0x01 & 0x03 = 0x01`
			`0xF0 & 0xAA = 0xA0`

OR	If either of the two inputs has a bit set, the output will as well.	|	`0x01 | 0x02 = 0x03`
			`0x01 | 0x03 = 0x03`
			`0xFF | 0x00 = 0xFF`

XOR	If only one of the two inputs has a bit set, the output will as well.	^	`0x01 ^ 0x02 = 0x03`
			`0x01 ^ 0x03 = 0x02`
			`0xAA ^ 0xF5 = 0x5F`

NOT	Every bit is set to its opposite.	~	`~0x01 = 0xFE`
			`~0x00 = 0xFF`
			`~0x55 = 0xAA`

```
test = register & bit;          // test a bit
test = register & (1 << 3);     // check 3rd bit
test = register & 0x08;         // same, different syntax

register = register | bit;      // set or turn on a bit
register = register | (1 << 3); // turn on the 3rd bit
register |= 0x08;               // same, different syntax

register = register ^ bit;       // toggle a bit using XOR
register = register ^ (1 << 3);  // toggle 3rd bit
register ^= 0x08;                // same, different syntax

register = register & ~bit;      // clear or turn off a bit
register = register & ~(1 << 3); // turn off the 3rd bit
register &= ~0x08;               // same, different syntax
```

Manipulating Bits

Binary	Hex	Decimal	Remember this number
0000	0	0	This one is easy.
0001	1	1	This is `(1 << 0)`.
0010	2	2	This is `(1 << 1)`. Shifting is the same as multiplying by $2^{shiftValue}$.
0011	3	3	Notice how in binary this is just the sum of one and two.
0100	4	4	`(1 << 2)` is a 1 shifted to the left by two zeros.
0101	5	5	This is an interesting number because every other bit is set.
0110	6	6	See how this looks like you could shift the three over to the left by one? This could be put together as `((1 << 2)|(1 << 1))`, or `((1 << 2) + (1 << 1))`, or, most commonly, `(3 << 1)`.
0111	7	7	Look at the pattern of binary bits. They are very repetitive. Learn the pattern, and you'll be able to generate this table if you need to.
1000	8	8	`(1 << 3)`. See how the shift and the number of zeros are related? If not, look at the binary representation of 2 and 4.
1001	9	9	We are about to go beyond the normal decimal numbers. Because there are more digits in hexadecimal, we'll borrow some from the alphabet. In the meantime, 9 is just 8 + 1.
1010	A	10	This is another special number with every other bit set.
1011	B	11	See how the last bit goes back and forth from 0 to 1? It signifies even and odd.
1100	C	12	Note how C is just 8 and 4 combined in binary? So of course it equals 12.
1101	D	13	The second bit from the right goes back and forth from 0 to 1 at half the speed of the first bit: 0, then 0, then 1, then 1, then repeat.
1110	E	14	The third bit also goes back and forth, but at half the rate of the second bit.
1111	F	15	All of the bits are set. This is an important one to remember.

See [Wikipedia Bitwise Operations]

`volatile` Keyword

A **`volatile`** variable may change asynchronously.

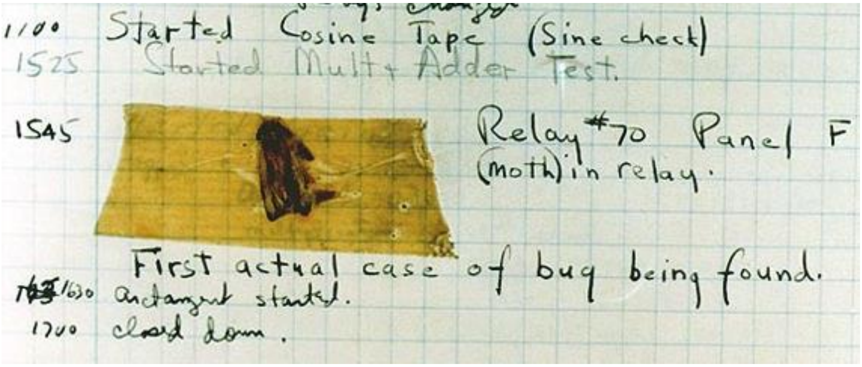
Used for registers and variables shared with interrupts.

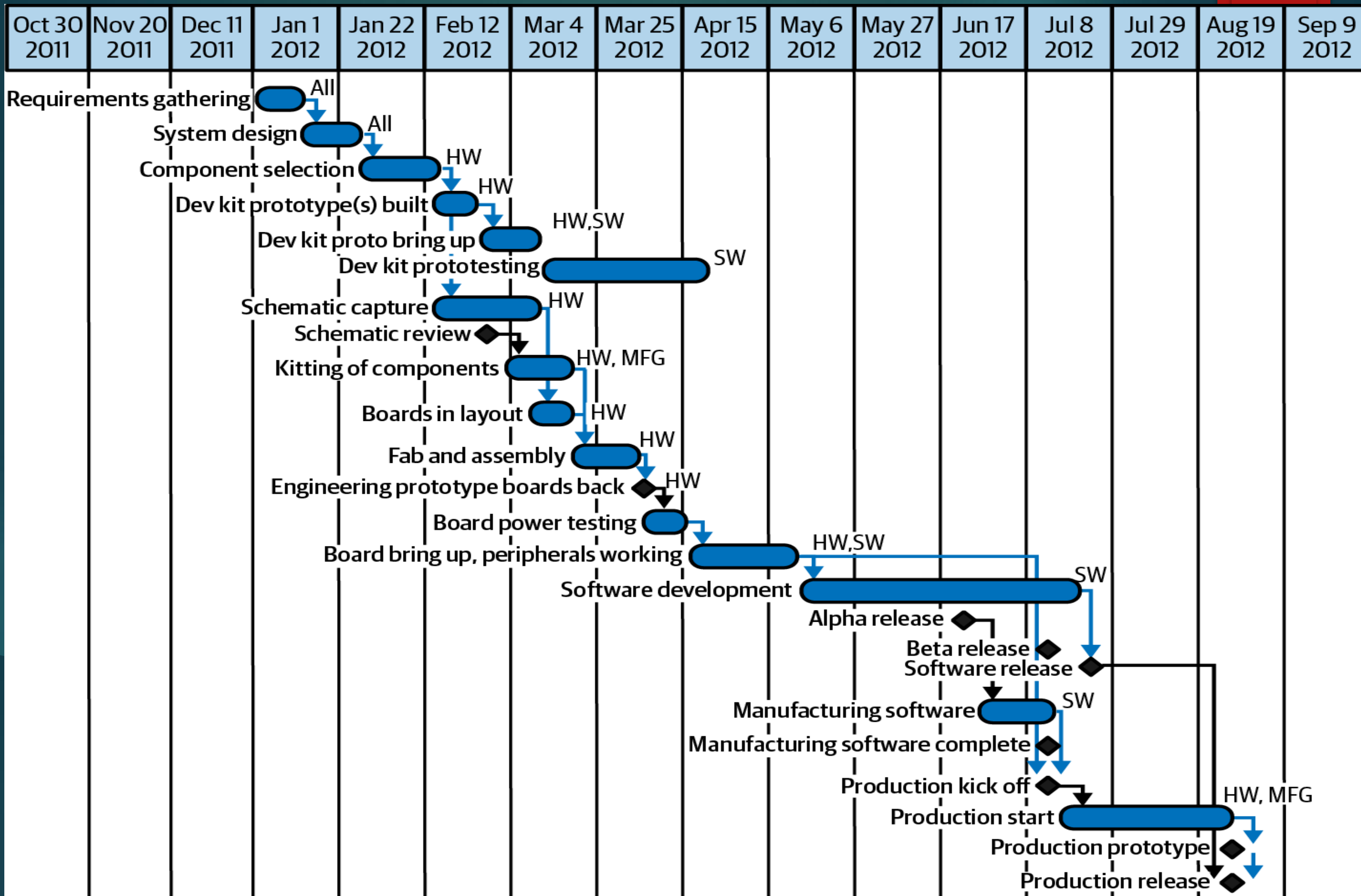
```
volatile uint32_t *reg;
reg = GPIO_REGISTER_ADDRESS;
*reg |= IO_RESET_LINE;  // set reset line high
DelayMs(50);
*reg &= ~IO_RESET_LINE; // set reset line low
```

```
volatile tBoolean gFeedChicken= FALSE; // global variable set by the interrupt
                                       // handler, which is cleared by normal
                                       // code when event handled
void TIM2_IRQHandler(void){
  __disable_irq();        // disallow nesting of interrupts
  gFeedChicken = TRUE;
  __enable_irq();
}
```

Software **and** Hardware Bugs

While Admiral Grace Hopper was working on a Mark II Computer at Harvard University in 1947, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system.

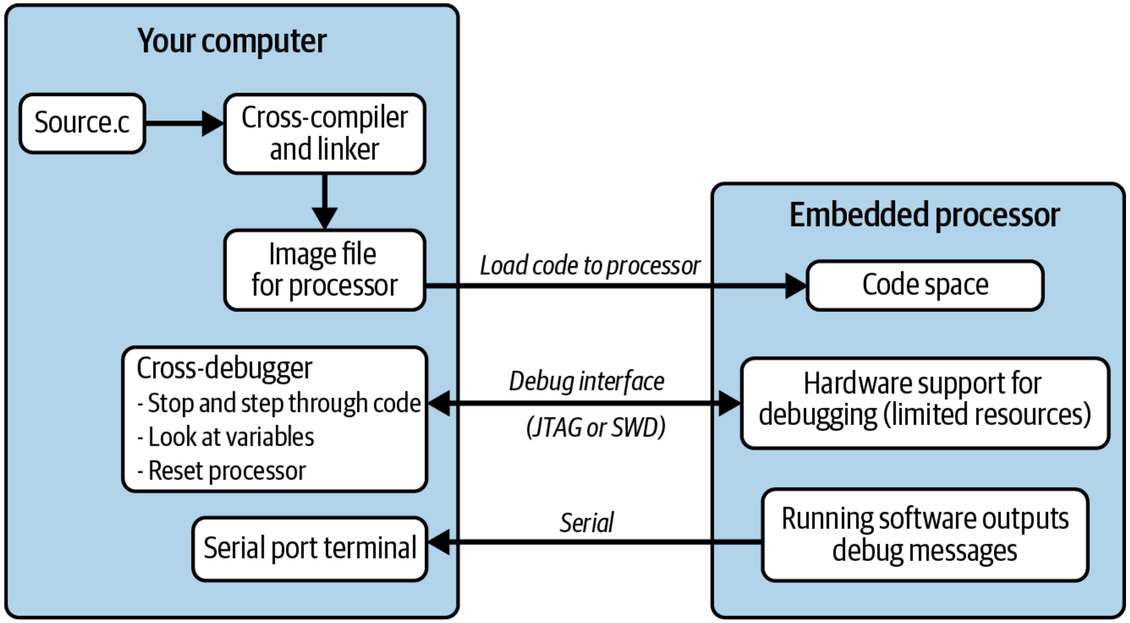


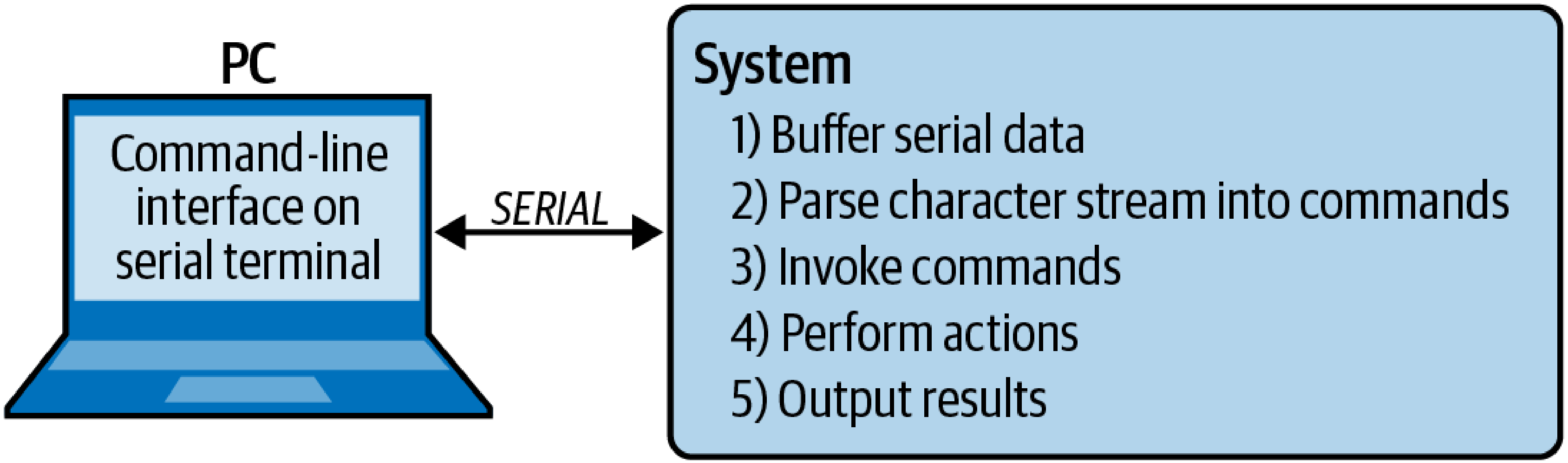


Scheduling Dependencies

The Device Is Not Your Computer

- **Cross-compiler** creates code for a processor different from what it is running on
- **Cross-debugger** (aka "jay-tag") debugs the remote processor
- **Serial ports** are generally used for logging and command line interfaces (CLIs)

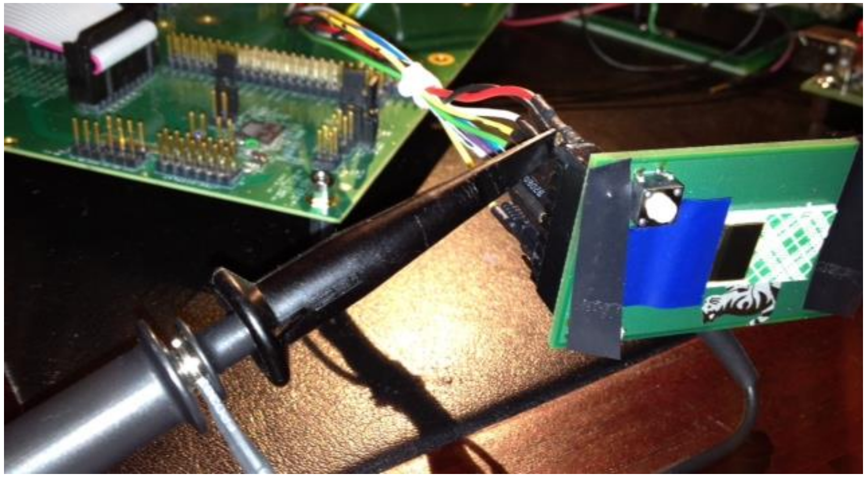


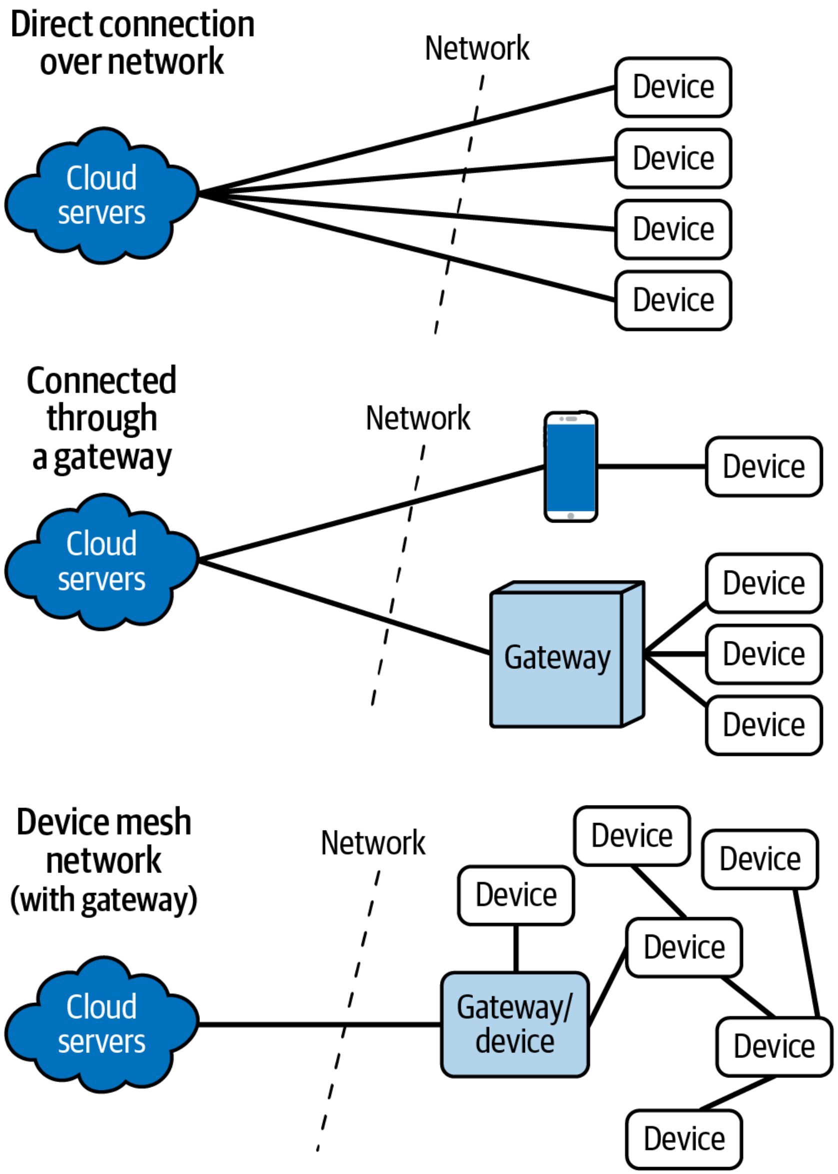


Debugging via `printf`

Testing Each Piece of the Device

- Application
- Processing
- Image collection
- SPI driver
- SPI wires
- ADC
- Sensor





Device Networks

Debugging the System

Is it powered? (Really?)

Is it running the code you think? (Really?)

Can you test only that part of the system?

Did you check the **errata** for the part?

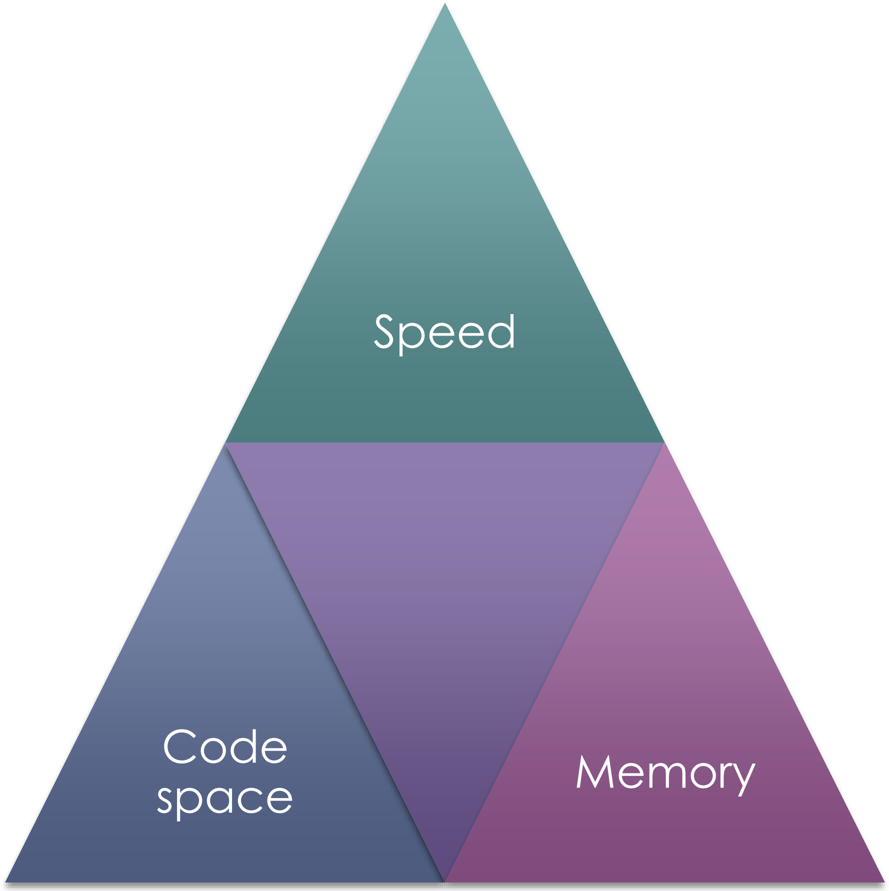
Have you described how the behavior is different from what you want?

Is it intermittent? Timing error | Uninitialized variable | Need volatile keyword | Stack problems

Can you turn **optimizations** off and see if it still happens?

Have you looked at the **map file**?

In case of emergency: ground loop? cosmic rays?



Resource Constraints

Optimization

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified" — Donald Knuth

"More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason — including blind stupidity." — W.A. Wulf

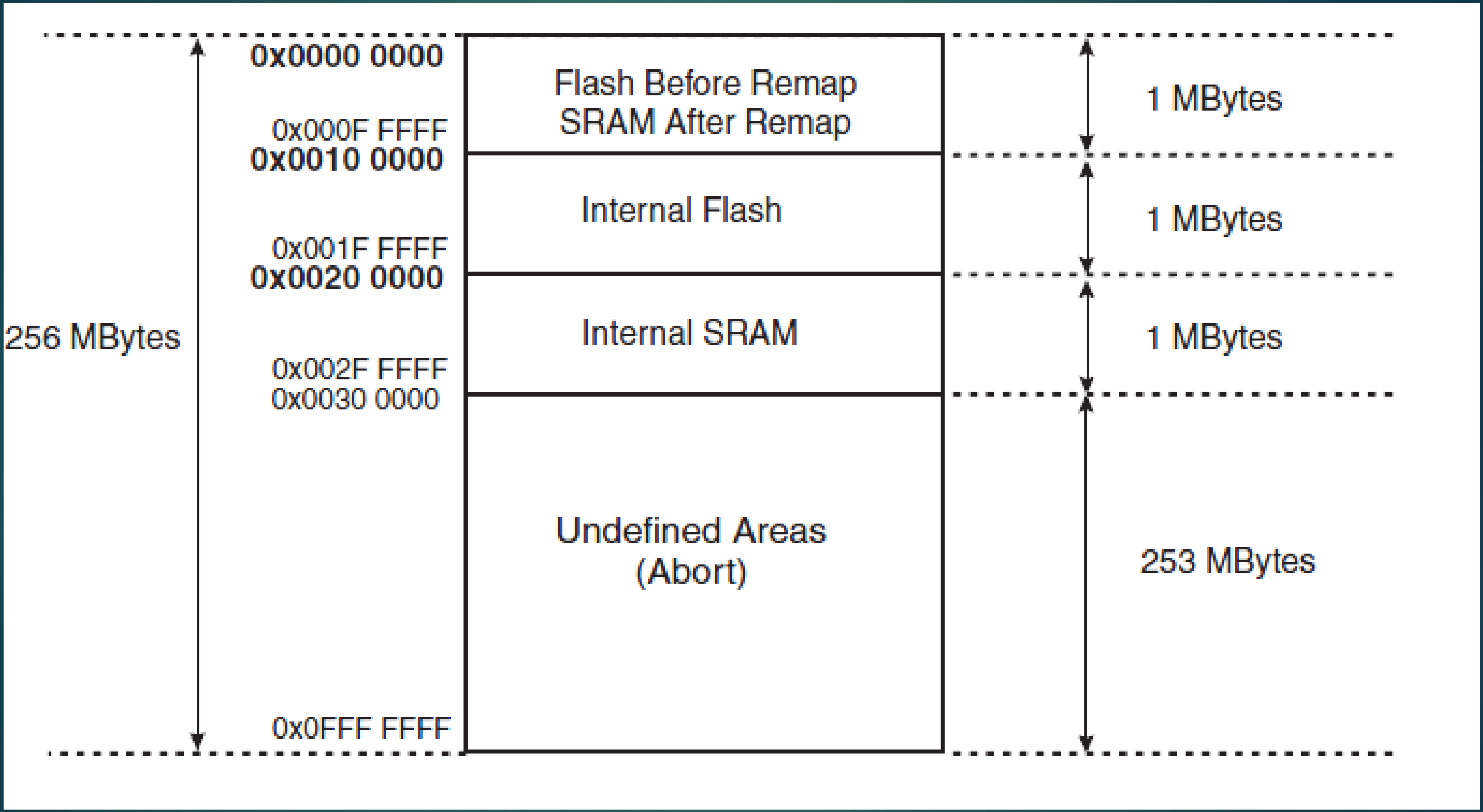
"Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you have proven that's where the bottleneck is." — Rob Pike

"The First Rule of Program Optimization: Don't do it. The Second Rule of Program Optimization (for experts only!): Don't do it yet." — Michael A. Jackson

Speed

- Power requirements
- Cost
- Required peripherals
- Processor
- Co-processor(s)
- Memory speed
- Non-processor memory transfers (DMA)
- Compiler optimizations

Memory (RAM)



0x0000 0000
0x000F FFFF
0x0010 0000
0x001F FFFF
0x0020 0000
0x002F FFFF
0x0030 0000
0x0FFF FFFF
256 MBytes
Flash Before Remap
SRAM After Remap
Internal Flash
Internal SRAM
Undefined Areas
(Abort)
1 MBytes
1 MBytes
1 MBytes
253 MBytes

```
Allocating common symbols
Common symbol       size              file
gNewFirmwareVersion
                    0x6               ./src/firmwareVersion.o
```

```
Memory Configuration
Name         Origin        Length       Attributes
Flash         0x00000000   0x00008000   xr
RAM           0x10000000   0x00002000   xrw
*default*     0x00000000   0xffffffff
```

```
.text.Initialize
                0x0000037c        0x7c ./src/main.o
                0x0000037c                 Initialize
 .text.main     0x000003f8        0xb4 ./src/main.o
                0x000003f8                 main
```

```
.section.functionName   address     size         file
                        address                  functionName
```

```
.text.__aeabi_ldiv0
  0x00006c6c  0x4    ../lib/gcc/arm-none-eabi/4.3.3/thumb2\libcr_eabihelpers.a(rtlib.o)
  0x00006c6c         __aeabi_ldiv0
.text.__bhs_ldivmod
  0x00006ec0  0x20c ../lib/gcc/arm-none-eabi/4.3.3/thumb2\libcr_eabihelpers.a(rtlib.o)
  0x00006ec0         __bhs_ldivmod
```

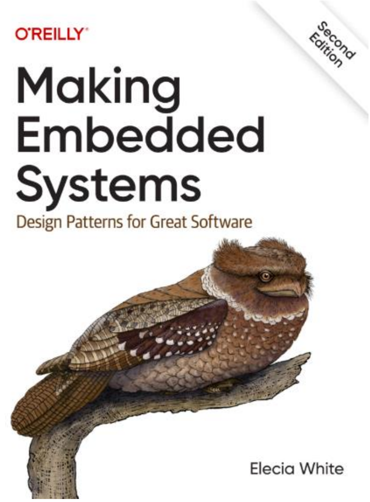
Code Space (ROM, Flash, MRAM)

Take-aways

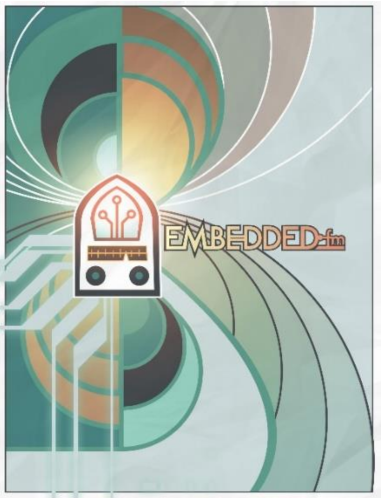
- Embedded systems involve hardware. It is a separate discipline from software and most people have big gaps in their knowledge.
- Processors are like languages, with their own finicky syntax (registers, bit manipulation, volatiles).
- Debugging an embedded system means identifying issues as hardware or software and then figuring out how to capture them. Designing for testability is critical.
- Hardware generally needs to be as cheap as possible which means software doesn't have as much RAM, code space and processing speed as it would like. Sometimes these resources can be traded for each other, but it makes embedded software fragile.

Take-aways

- Embedded systems involve hardware. It is a separate discipline from software and most people have big gaps in their knowledge.
- Processors are like languages, with their own finicky syntax (registers, bit manipulation, volatiles).
- Debugging an embedded system means identifying issues as hardware or software and then figuring out how to capture them. Designing for testability is critical.
- Hardware generally needs to be as cheap as possible which means software doesn't have as much RAM, code space and processing speed as it would like. Sometimes these resources can be traded for each other, but it makes embedded software fragile.



O'REILLY
Second Edition
Making Embedded Systems
Design Patterns for Great Software
Elecia White



Thank you!

Bonus Slides

What else do you want to know?

Resources

Making Embedded Systems, 2nd Edition

github.com/eleciawhite/making-embedded-systems Slides for this talk are in the Presentations folder.

Embedded.fm is a podcast about engineering, art, education, and technology.

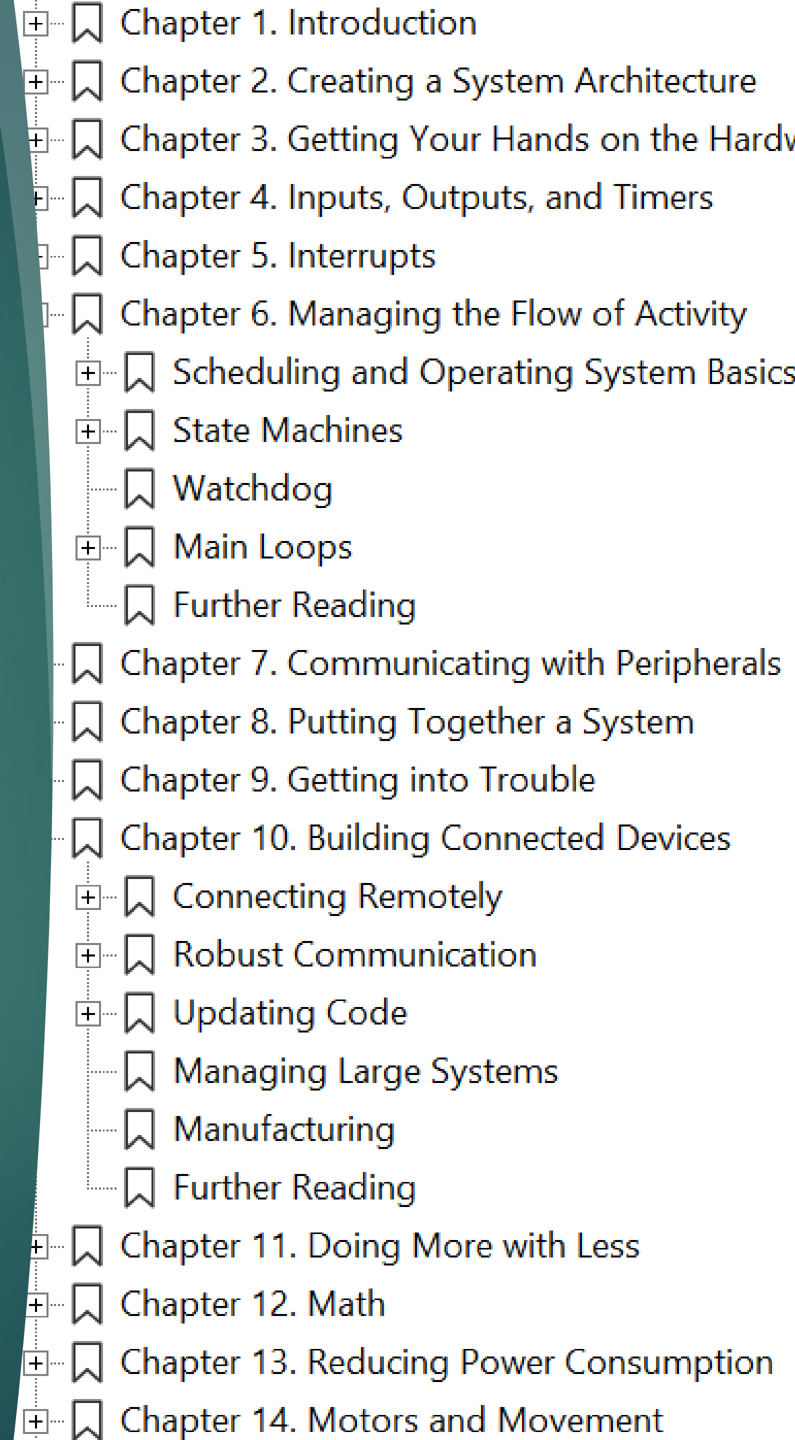
Buried Treasure and Map Files is my presentation about memory map files.

O'Reilly Learning Platform:

30-day trial

What’s New in the 2nd Edition?

- New chapters include:
  - Interrupts
    - There is this whole thing with a chicken pressing a button
  - Managing the Flow of Activity
    - How to set up your main loop
  - Getting into Trouble (debugging)
  - Motors and Movement
- Architecture diagrams section overhaul
- Bootloaders changed to covering whole IoT systems (Building Connected Device)
- Updated information (more HALs)
- New figures!



Learning platform

Give everyone in your organization what they need to solve problems and drive productivity

- Expertly curated, high-quality content
- Multiple learning formats for different learner types
- Solve real-life problems in real-time
- Hands-on labs and sandboxes (Learn to do)
- Deep insights dashboard
- Live online courses