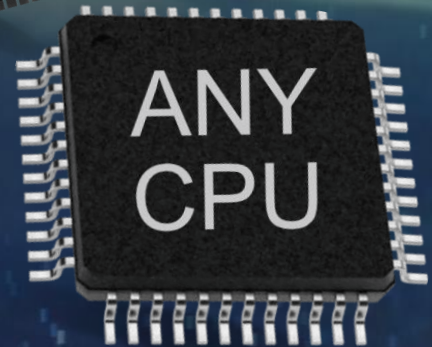
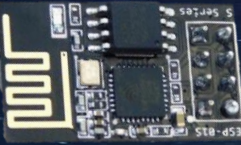


Write Embedded C Faster, Better, Safer With **Fin**

AN INTRODUCTION

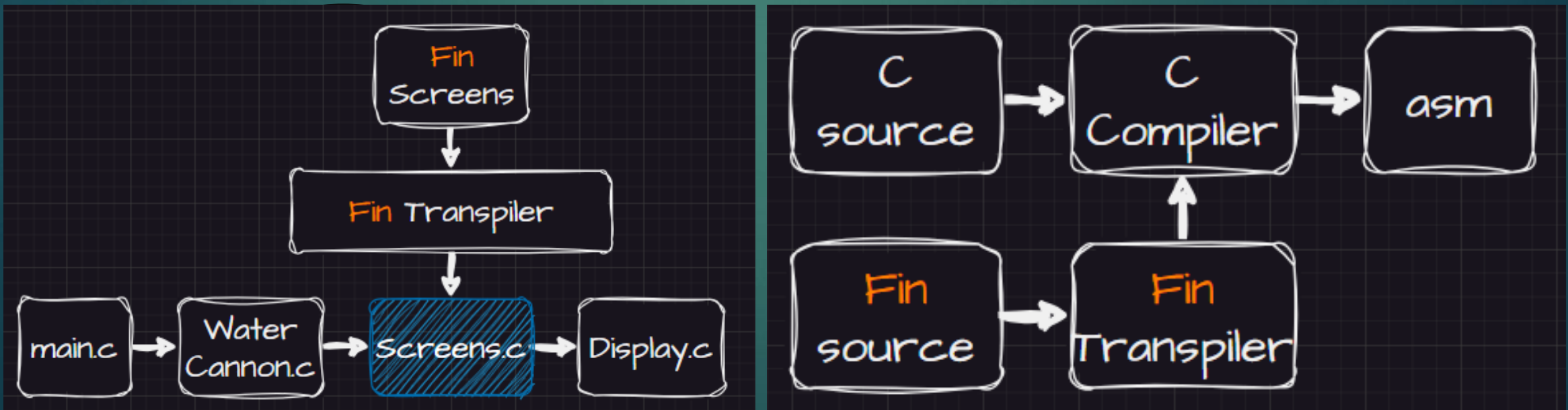


Fin does NOT replace C

Unlike most new languages, fin does not aim to replace C/C++

Instead, fin transpiles to human readable C99. No C barf. Near 1-to-1 translation.

The generated C code works with existing projects - no heap, or GC required



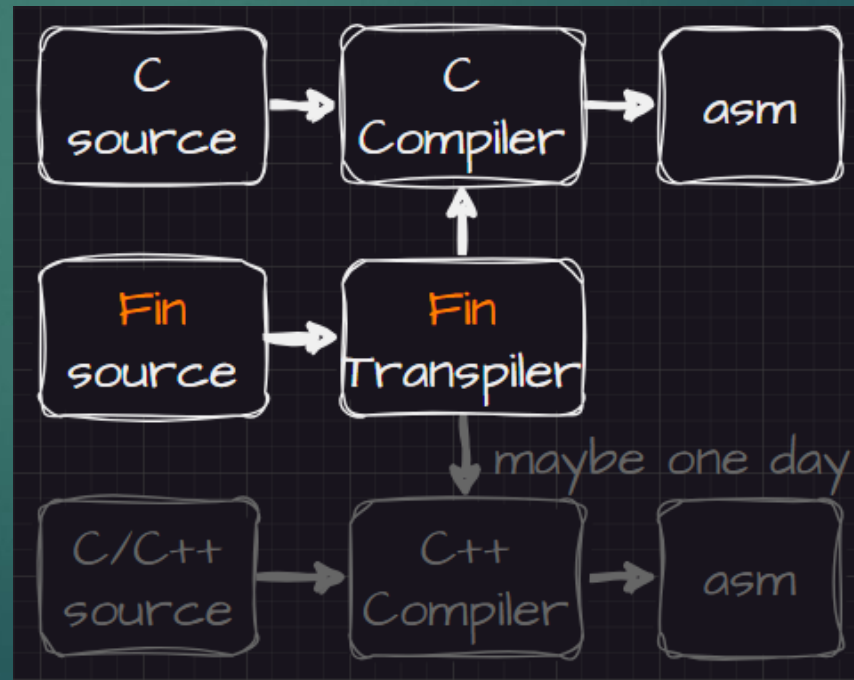
Fin ♡ C

I actually really like C - simplicity, efficiency, control over memory and resources
There's a C compiler/debugger for every embedded architecture - **important!**

Fin is heavily based on C, but is also a higher level language with many benefits.
Kinda like a modern C++ that is

- ▶ much simpler
- ▶ way safer
- ▶ with superb tooling

Maybe one day fin will also transpile to C++, but the primary goal is to target C.



Goals

Fin makes it **easy** for us to **quickly** write C code that is

Modular, yet Efficient

- ▶ In C, these goals often conflict
- ▶ **No heap** or GC required



Easy to Test and Simulate

- ▶ C testing is usually not fun
- ▶ Python like test abilities



Safer, yet Familiar

- ▶ Solves many safety issues
- ▶ Productive from day 1

The fin programming language

This “new” programming language is very similar to C.

Shares much (but not all) of C syntax.

No wild paradigm shifts.

Easy to learn.

Valid C99 syntax

Valid **fin** syntax

```
float calc_stuff(int a, float b) {  
    return a * b + get_offset();  
}
```



Experimental

Fin is experimental.

What happens if the experiment fails?

You'll be left with quality human readable generated C99 code that mirrors your **fin** code.

Curb your enthusiasm darn it!

- ▶ many fin concepts have been validated in prototypes
- ▶ no initial compiler release yet
- ▶ looking for feedback on direction

Quality Readable C99

Fin generates high quality C99 code that follows your input.

fin code

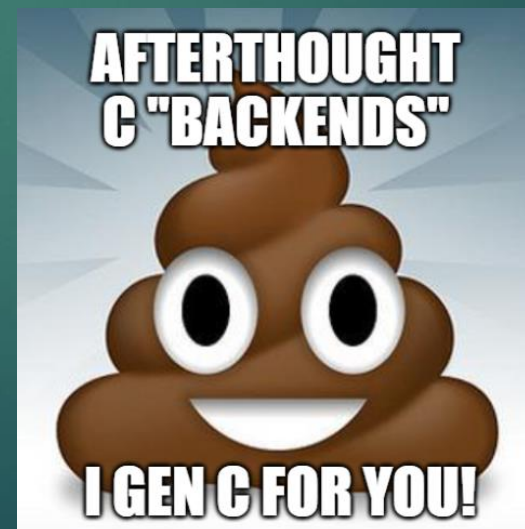
```
// Some user comment about Car
public class Car : Finobj
{
    // user comment about count
    uint count = 5;
    int speed; // m/s
}
```

Generated C99

```
// Some user comment about Car
typedef struct Car
{
    // user comment about count
    uint32_t count;
    int32_t speed; // m/s
} Car;
```

This is a primary goal. Not an afterthought.

Fin generated code doesn't look like it has been digested and plopped out the backend of a compiler.

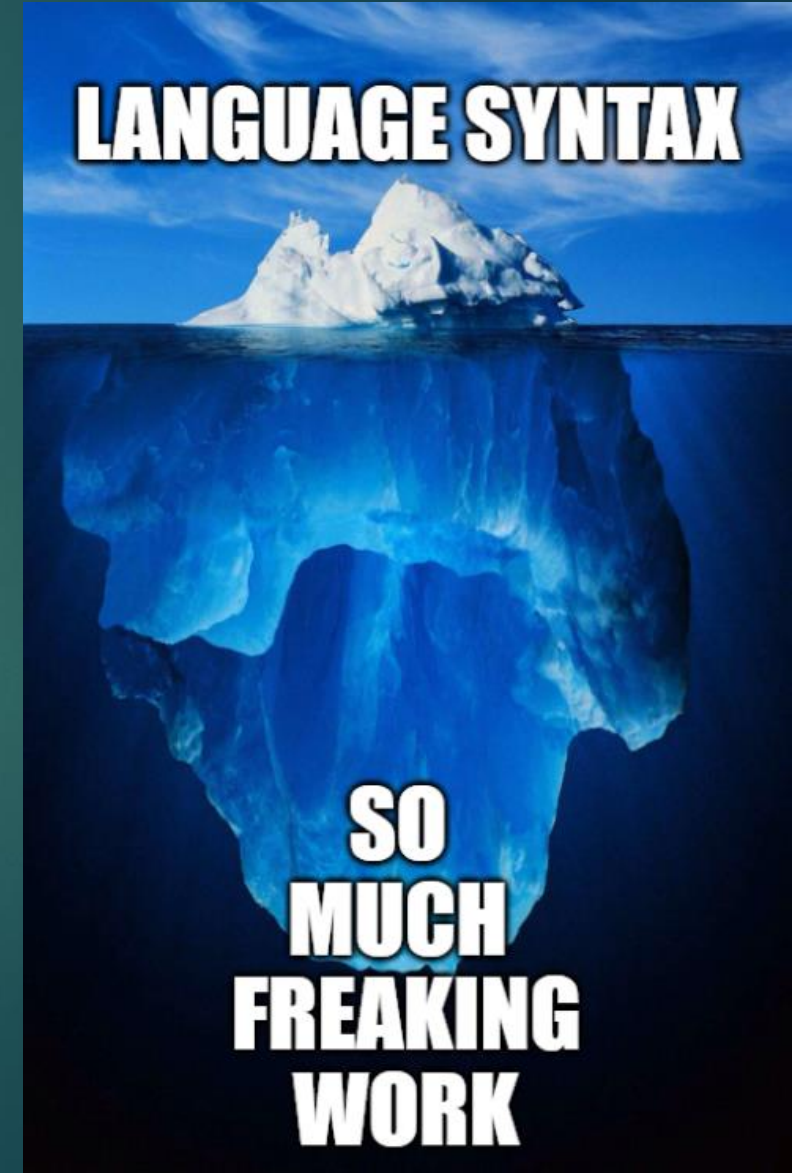


TLDR – **fin** is a subset of C#

Fin is basically a **subset** of C# suitable for embedded.

Flippant reasons to base fin on C#:

- ▶ It's close enough to what I want &
- ▶ I'm lazy
 - ▶ Tons of work for IDE, debugger, package mgmt...
- ▶ I want it to be usable in a couple months
- ▶ The world doesn't need another unique programming language syntax (for my goals)



Getting Bored?

Video 1/6 in series exploring embedded system challenges with C and how fin helps us. See description for timestamps and links. Jump, jump around.



Write Embedded C
Faster, Better, Safer
With **Fin**

AN INTRODUCTION



Fin Language
Features

SOME OF THE GOOD STUFF



Multiple Controller
Simulations FTW

JAVA TO C99 (10 YEARS AGO)



Embedded C
Software Testing
& Design Choices

WITH ASSOCIATED CHALLENGES



Language
Comparison



NO LANGUAGES WERE HURT...

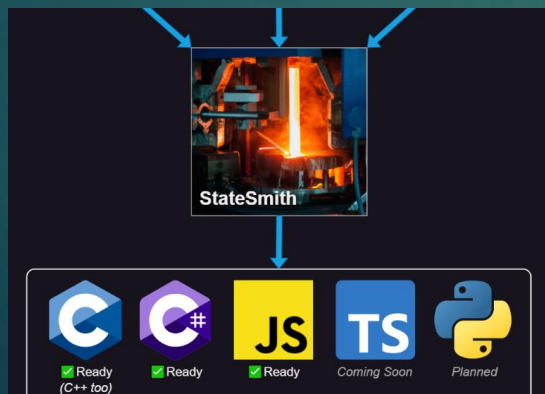


Fin Implementation
And Challenges

WE CAN DO IT!

Detailed Reasons For C#

- ▶ C# compiler (Roslyn) open source, vscode C# open source. 
- ▶ Core C# is close to C/C++.
- ▶ Fin will **NOT** support all C# syntax or lib (available for testing and simulation though).
- ▶ Unit testing, simulating, mocking in C# is fast, easy and very powerful.
- ▶ Well known syntax. Good for AI helpers, lots of tutorials.
- ▶ Incredible debugging (hot code reload, evaluation window). 
- ▶ Solid tooling. Fearless refactoring. Auto code analysis and fixes.
- ▶ **Transpiling is easier with C# Roslyn. I already do simple C# to C or JS for StateSmith.**



Testing With Fin

- ▶ C# has many excellent mocking libraries!
- ▶ Amazingly simple to replace a module with an auto mock. Yep! AUTO mock :)

fin code (C#)

```
public class Screens
{
    Display _display;

    public Screens(Display display)
    {
        _display = display;
    }

    public void show_idle()
    {
        _display.set_title("Idle");
        _display.set_subtitle("Press OK");
    }
}
```

Test/simulation code (C#)

```
public class ScreensTest
{
    [Fact]
    public void ShowIdle()
    {
        // arrange
        var mock_display = Substitute.For<Display>();
        var screens = new Screens(mock_display);

        // act
        screens.show_idle();

        // assert expected calls received by mock
        mock_display.Received().set_title("Idle");
        mock_display.Received().set_subtitle("Press OK");
    }
}
```

Testing Compared

- ▶ Don't take the previous fin test example for granted.
- ▶ It's generally a lot more work to do in C.
- ▶ We didn't have to mix C with C++ for efficient unit testing.
- ▶ We didn't have to setup and maintain multiple test programs.
- ▶ We didn't have to hand write a C++ mock interface and class using gtest macros

```
class IDisplay {  
    public:  
    virtual ~IDisplay() {}  
    virtual void set_title(const char* title) = 0;  
    virtual void set_subtitle(const char* subtitle) = 0;  
};
```

```
class MockDisplay : public IDisplay {  
    public:  
    //...  
    MOCK_METHOD(void, set_title, (const char* title), (override));  
    MOCK_METHOD(void, set_subtitle, (const char* subtitle), (override));  
    //...  
};
```

```
// PLUS MORE code to link C calls  
// to C++ mocks
```

```
// PLUS actual test code...
```

Testing Compared

- ▶ We didn't have to run custom mock generator tools (can be bad for refactoring).
- ▶ We didn't have to settle for brittle/primitive C mocking library capabilities.
- ▶ We didn't have to remember to add our tests to a runner list.

- ▶ We got a **best-in-class experience** with near **zero effort**.
- ▶ Just a single call `Substitute.For<Display>();`
- ▶ Our tests are easily run in parallel to significantly speed up testing. **10x?**
- ▶ We can write our test code in the same file as the implementation if we want.

Rock the mock like python

- ▶ Regular C# mocking usually requires an interface like `IDisplay` to work.
- ▶ Coding against interfaces can be a good practice, but it is also extra work.
- ▶ Fin aims to support easy testing however the developer wants to code.
- ▶ Test/simulation code can access anything (or not if you like).
 - ▶ Want to reach in and test a private method? Go ahead. Nothing special required.
 - ▶ Hate that? OK. Don't do it.
 - ▶ Really hate that? OK. Write an analyzer for your group to prevent it.
 - ▶ You do you.
- ▶ Private/protected still respected for non-test code.
 - ▶ This probably needs an example. More on this later.



Interface when you want

Use of interfaces helps make code re-usable, but it also makes it harder for users to follow code.

Navigating without interfaces

▶ WaterCannon -> Screens -> Display

Navigating with interfaces

▶ WaterCannon -> IScreens -> find implementers (Screens) -> IDisplay -> find implementers (Display)

If target/production code doesn't need an interface for runtime polymorphism, you **won't need** to add one just for easy testing.

Pound out an implementation to get it working, add interfaces if/when needed. Fin makes this easy.

Generated Code For Test Example

Fin realizes it should generate single instances of Screens and Display modules. It will generate **efficient** C code from **highly modular** and **testable** fin code.

fin code (C#)

```
public class Screens
{
    Display _display;

    public Screens(Display display)
    {
        _display = display;
    }

    public void show_idle()
    {
        _display.set_title("Idle");
        _display.set_subtitle("Press OK");
    }
}
```

Screens.h

```
#pragma once // or include guard
void Screens_show_idle(void);
```

Screens.c

```
#include "Screens.h"
void Screens_show_idle(void)
{
    Display_set_title("Idle");
    Display_set_subtitle("Press OK");
}
```


Choose 3

No need to compromise on things that really matter.

✓ **Modular, yet Efficient**

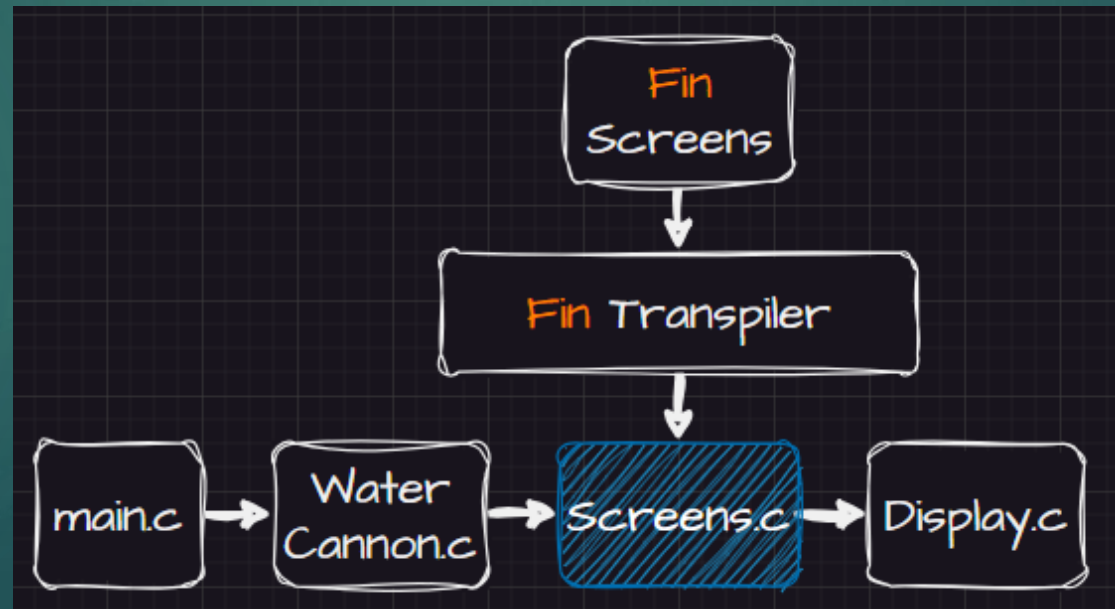
✓ **Easy to Test and Simulate**

✓ **Safer, yet Familiar**

- ▶ Example didn't really show safety features
- ▶ Fin removes need for most null checks

Deploying fin – existing code

- ▶ You can mix and match fin modules with regular C modules.
- ▶ Fin generated .c can call handwritten .c and vice versa.
- ▶ You can use fin to add new modules to an existing code base,
- ▶ Or convert modules to fin one at a time,
- ▶ Or write an entire embedded application in fin.



Still Interested?

Video 2/6 is up next... See description for links.

Write Embedded C
Faster, Better, Safer
With Fin

AN INTRODUCTION



Fin Language
Features

SOME OF THE GOOD STUFF



Multiple Controller
Simulations FTW

JAVA TO C99 (10 YEARS AGO)



Embedded C
Software Testing
& Design Choices

WITH ASSOCIATED CHALLENGES

Language
Comparison

NO LANGUAGES WERE HURT...



Fin Implementation
And Challenges

WE CAN DO IT!



Fin Language Features

SOME OF THE GOOD STUFF



Some Features Covered

- ▶ No more header files!
- ▶ Null stuff
 - ▶ Non-null types
 - ▶ Prevent pass by copy mistakes
 - ▶ Null analysis
- ▶ Function stuff
 - ▶ Out parameters
 - ▶ Tuples
 - ▶ Named arguments
- ▶ Simple generics
- ▶ Safety stuff
 - ▶ strings
 - ▶ arrays
 - ▶ error handling
 - ▶ arithmetic, overflows
 - ▶ non blocking assurance
 - ▶ data thread safety
- ▶ Selective safety escapes
- ▶ Support code generation
- ▶ Portable de/serialization

Write It Once

No more writing function prototypes and flipping header files!

Write your code once and move on.

No need to worry or question about keeping .h and .c documentation in sync.

```
// fin
// some documentation
public void send(Packet packet) {
    // ...
}
```

```
// C99 - SomeClass.h
// packet should not be null
// some documentation...
void SomeClass_send(Packet *packet);
```

```
// C99 - SomeClass.c
// packet should not be null
// some documentation...
void SomeClass_send(Packet *packet) {
    // ...
}
```

No Null Checks Needed

A fin object reference is essentially a C pointer that can't be null.

Similar to C++ references.

C's pointer ambiguity is a common source of errors or unnecessary error checking.

Every time you write a C function that takes a pointer, you need to decide:

- ▶ Is a comment explaining nullability needed? Comments also often rot.
- ▶ Is a null check required? What do we do if an unexpected null is passed in?

```
// fin
public void send(Packet packet)
{
    // No null checks needed :)
    uart.send(packet);
}
```

```
// C99
// packet should not be null
void SomeClass_send(Packet *packet)
{
    if (packet == NULL) { // needed?
        return; // TODO error?
    }
    uart.send(packet);
}
```

Pass Objects by Reference

C - passing a struct by copy is a common mistake.

► C++ **very common** mistake. Is it pass by ref or copy? Need to check function.

Fin - pass objects by reference, NOT by copy. Primitives (ints) by copy still.

Fin - copying of memory is explicit. No accidental copying of large structures.

```
// fin
public void send(mem<Packet> packet)
{
    //...
}

//using code REQUIRES explicit copy
send(mem.copy(packet)); // mem.cp?
send(packet.copy());
```

```
//using code
send(packet); // passed by copy? Ref?
```

```
// C/C++
void send(Packet packet)
{
    // did we mean to get a copy?
}
```


Null Analysis

Fin - if a passed parameter can be null, you must check before use.

```
public void maybe_send(Packet? packet)
{
    uart.send(packet);
}
```

(parameter) Packet? packet

'packet' may be null here.

CS8604: Possible null reference argument for parameter 'packet' in 'void uart.send(Packet packet)'.

No compiler error.

```
public void maybe_send(Packet? packet)
{
    if (packet != null)
        uart.send(packet);
}
```

Ignore null warning with “dammit” operator.

```
public void maybe_send(Packet? packet)
{
    uart.send(packet!);
}
```

Out Parameters

The out keyword is better than using comments in C. Tuples are good too.

```
// fin
public void divide(int number, int divisor, out int quotient, out int remainder) {
    quotient = number / divisor;
    remainder = number % divisor;
}
int quotient;
divide(10, 4, out quotient, out var remainder); // note use of `out var`
```

```
/**
 * C ...
 * @param quotient [out] must not be null
 * @param remainder [out] must not be null
 */
void divide(int number, int divisor, /* out */ int *quotient, /* out */ int *remainder) {
    *quotient = number / divisor;
    *remainder = number % divisor;
}
int quotient; int remainder;
divide(10, 4, &quotient, &remainder);
```

Multiple Returned Values With Tuple

Tuples are a convenience. You can return an object if you want.

```
// fin
public (int quotient, int remainder) divide2(int number, int divisor)
{
    int quotient = number / divisor;
    int remainder = number % divisor;
    return (quotient, remainder); // can specify field names too
}

var result = divide2(10, 4);
// result.quotient
// result.remainder

var (quotient, remainder) = divide2(10, 4);
```

Named Arguments

Very helpful for functions that take multiple args with the same type.

Note in below example, order of named arguments doesn't need to match signature.

Very handy when working with libs that use ordering different conventions.

This simple feature can aid in readability and prevent bugs.

```
// fin
public double calc_stuff(double x, double y, double width, double height)
{
    // stuff
}

double stuff = calc_stuff(x: 2.0, width: 10.0, y: 3.0, height: 5.0);
```

Simple Generics

C has no good way of reusing containers like Queues for different data types.

Most people I know just copy paste `IntQueue` and change into `ByteQueue`.

Fine until you find a bug or want to make a change. Fix `ByteQueue`, forget `IntQueue`...

I've abused preprocessor to get around this. Interesting, but clunky. Not ideal.

Other suboptimal solutions (heap with void pointers, unions...)

Simple in fin.

```
// fin
public class Queue<T> { // T can be int, struct...
    // stuff
}
```

Fin/C# generics are not as powerful as C++ templates, but also much nicer to use.

C++ templates lead to horrific compile errors, result in duck typing, break refactoring...

My old prof had a comic about C++ on his door. Write 1 line. Compile. Repeat.

Safer Strings

C strings are a very common source of bugs.

Fin will support C style null terminated strings, but will also add safer strings.

Initially, fin string lib will focus only on non-heap string functionality.



Safer Arrays

C memory safety is always a concern.

Fin will support C style naked arrays, but will add safer arrays that track their own length.

Open question about how to handle out of bounds array access:

- ▶ Panic?
- ▶ Ignore in certain builds?
- ▶ Throw lightweight exception?

One day will support fixed sized arrays like C++ `std::array<int, 5>`.

Potential syntax ideas:

- ▶ `array5<int> arr_of_5_ints;`
- ▶ `array<int, L5> arr_of_5_ints;`

Not as nice as C's syntax, but then C messed it up too.
Google "C's biggest mistake".



Future - Arithmetic Overflow

Overflow is a common annoyance for programmers and code reviews.

Add special types that throw on overflow. Default?

Add special types that don't care/check for overflow.

Add special types that saturate.

Future – support static analysis and hints that ensure no overflow at compile time.

Learn from existing approaches:

- ▶ <https://github.com/dcleblanc/SafeInt>
- ▶ <https://github.com/davidstone/bounded-integer> (featured on CppCast)
 - ▶ Criticisms <https://news.ycombinator.com/item?id=21107072>
 - ▶ "oh good, my compile times aren't long enough, lets make EVERY INTEGER a template"
 - ▶ Perhaps checks run only when requested by user. Keep development cycle fast.



Future – Non-Blocking Assurance

Interrupts should be short and sweet, but sometimes you call functions.

It would be really nice if `fin` could check all paths and ensure none of them block.

I've seen cases where a rare error branch is taken and a debug msg is printed, but writer didn't anticipate being called by an IRQ. Oops! It works 99% of the time fine :)

```
// fin
[non_blocking]
public void gpio_irq()
{
    // some function calls...
}
```



Future – Critical Section Audits

Critical sections should be short and sweet (like an interrupt), but yeah, sometimes we call functions or do some processing.

It would be helpful to ensure all code paths inside a critical section **cannot** block or call any functions (like OS functions) which will cause a problem.

*“FreeRTOS API functions **must not** be called from within a critical section.”*



Future – Data Thread Safety

Not sure how I would implement yet, but I'd like fin to be able to detect data concurrency issues.

Consider threads and interrupts.

Could use annotations for simple cases.

Harder if a single class is instantiated multiple times for multiple threads.

For hard cases, could detect during simulation/testing if data was accessed by multiple threads outside of a safe/critical section.



Future – Safety Escapes

Safety will be the default, but sometimes it really isn't needed.

Fin will make it easy to enable/disable certain checks in code without needing to modify the actual source code. Hopefully even for libraries!

More on this later.



Future - Lightweight Exceptions?

Belay the pitchforks! 

There is a lot of dislike towards exceptions as implemented in other languages:

- ▶ Hidden flow control – this is a big one. Hard to audit/understand code.
- ▶ C++ code size - exception tables and unwind data can be large.
- ▶ Performance cost – throwing an exception can be expensive.
- ▶ Java checked exceptions annoying – often circumvented.

Explore exceptions implemented with return codes in C. Not heavy like C++.

Fin lib will support exceptions AND traditional bools/error codes (better with tuples).

Let users choose the appropriate tool for the job at hand.



Future - Lightweight Exceptions?

How to address hidden flow control? Flow control visible in generated C.

`mt__` prefix required for any function that may throw. Thoughts? Suggestions?

```
// fin
int id = mt__get_id(packet);
var response = mt__get_response_from_id(id);
response.mt__send();
```

Exceptions/sticky errors are really handy when you want to do bulk error handling instead of having a ton of if/else if/else if error handling at each possible fail point.

What if a deeply nested function like `d()` adds functionality that might throw?

`a()` calls `b()` which calls `c()` which calls `d()`

You either handle the exception (try/catch) in `d()` or prefix and let exception bubble.

```
mt__a(); mt__b(); mt__c(); mt__d();
```

Fin will add tooling to make these renamings easy (a few seconds work).

Experimental. Needs use to see if we like it.

Future – Support Code Gen

Even though fin will make writing C code more enjoyable, some projects still require a ton of boilerplate code (like wiring up 300 controller parameters).

Programmers are not data entry specialists!

Or copy paste cowboys.

We are thinkers, problem solvers, LED junkies.

Instead of resorting to cryptic powerful macros/templates and arcane incantations, I would rather write or use some simple code generation scripts where I can see the output.

Fin will make it easy for users to code gen more fin code.



Future - Portable De/Serialization

No more relying on specific bitfield, struct layouts, padding...

Most developers know that we shouldn't memcpy() packets, but do it anyway because the alternative is labor intensive.

Fin library will make this easy. Something like:

```
// fin
bool success = SomeFinLib.try_deserialize(to: packet, from: bytes);
```

```
// fin
using var packet = SomeFinLib.mt__deserialize<Packet>(bytes);
```



Modular And Efficient Code

Later videos show how fin also allows you to only pay for modularity when you actually need it at runtime (not just for testing).

- ▶ Single or multi instance
- ▶ Auto implemented vtables (only when actually needed)



Still Interested?

Video 3/6 is up next... See description for links.

Write Embedded C
Faster, Better, Safer
With Fin

AN INTRODUCTION



Fin Language
Features

SOME OF THE GOOD STUFF



Multiple Controller
Simulations FTW

JAVA TO C99 (10 YEARS AGO)



Embedded C
Software Testing
& Design Choices

WITH ASSOCIATED CHALLENGES

Language
Comparison

NO LANGUAGES WERE HURT...



Fin Implementation
And Challenges

WE CAN DO IT!





Multiple Controller Simulations FTW

JAVA TO C99 (10 YEARS AGO)

Story Time – Java to C

I created a Java to C99 transpiler about 10 years ago.

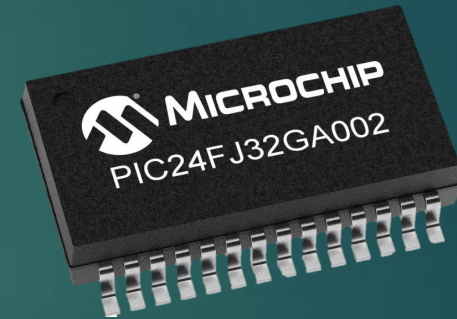
Generated C code used for a commercial product. No bugs found in field.



Highly testable,
modular code



Highly efficient
code. No heap.



MCU

Story Time – My History

Why the PIC24FJ32? Some background...

Lucky to start electronics consulting in high school (2002) for 2 different entrepreneurs.

- ▶ Chris – camera solutions for agriculture. Lived in Brandon Manitoba.
- ▶ George – paintball field operator in another province.



Highschool 2003
PIC16F (assembly)
Paintball Wireless
Basemaster System

George was a character :)
More basemaster stories
another day...



Story Time – Atmel AVR LOVE

I didn't learn C in high school so I did everything in assembly.

Switching from PIC16 assembly to Atmel AVR assembly (~2004) was bliss!

*“Overall, I've written some asm for both processors and I hate to break it to PIC users but writing assembly for PIC is akin to **stabbing myself in the face**.*

*Except its not even that efficient, cause you have to move the knife into the working register first (**movlw KNIFE**), and then you can stab yourself (**movwf FACE**).”*

<http://www.ladyada.net/library/picvsavr.html>



Story Time – My Mistake

While in university (2005), I designed an Expansion Camera product for Chris. It was our 3rd or 4th product and we were looking to keep part costs low.

I selected the ATtiny2313.

Boy did I screw myself. 😏

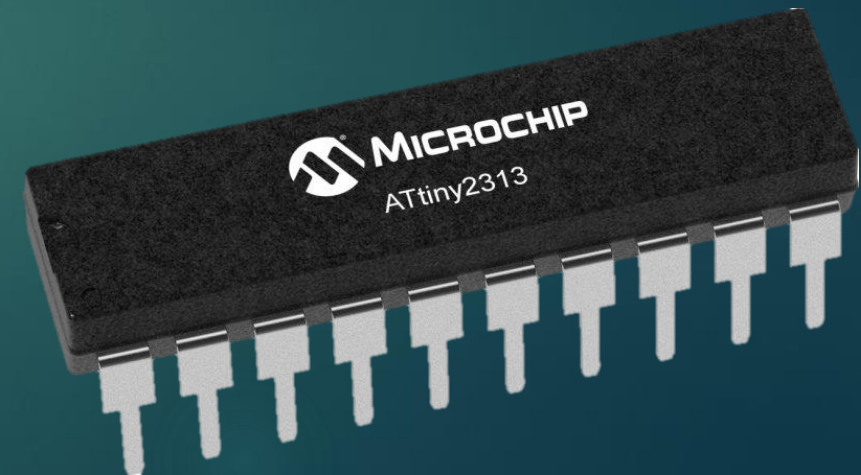
Super tight on RAM (128 Bytes).

Bit banged a communication interface.

Any savings on parts was negated by effort (selling hundreds not thousands).

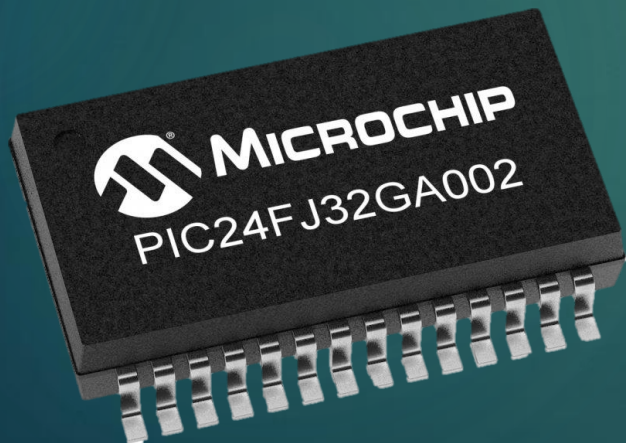
It worked well, and was interesting, but a lot of work.

Haven't made that mistake again :)



Story Time – Lessons Learned

While working for a telco (post computer engineering degree) I took a Chris side job. The Chris work never did pay great, but I sure learned a lot from the experience! I knew I needed 2 UARTs and my bit banging days for commercial work were over. I once created an ATtiny85 NTSC video character display (20x20, shading, borders...). The mPTZ had to be small and at the time, only large AVR ICs had 2 UARTs. The PIC24FJ32GA002 had a free C compiler, 2 UARTS, and enough RAM for buffering UARTs.

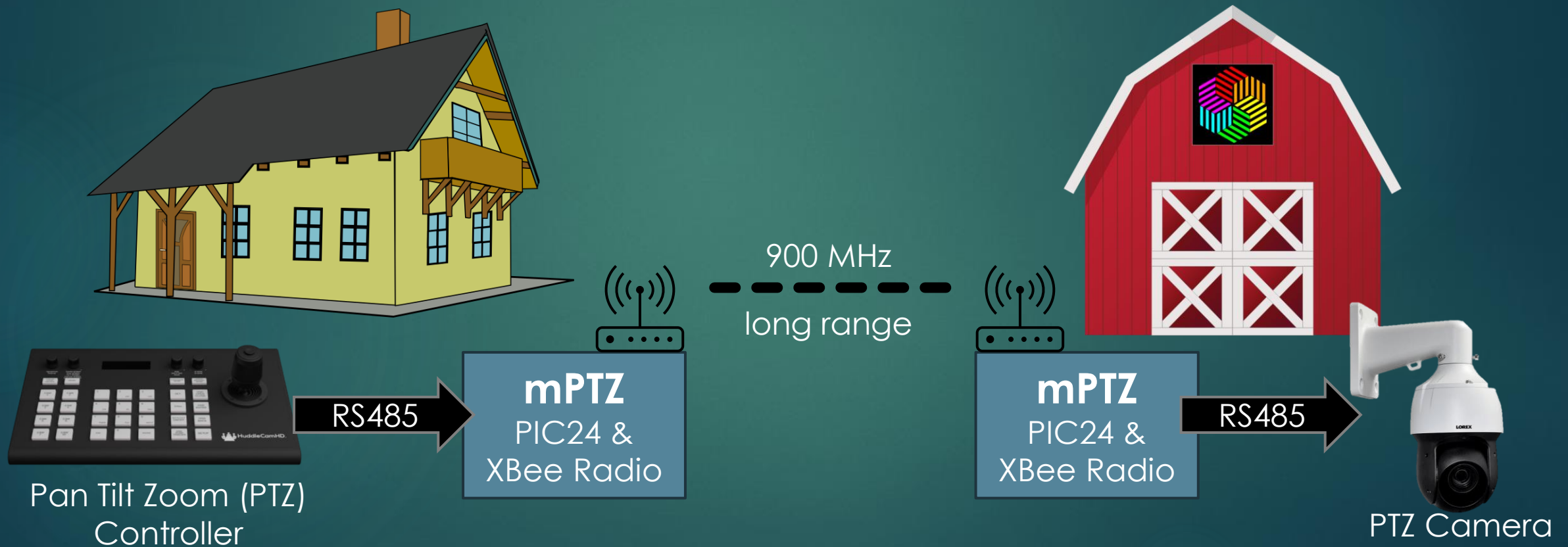


Core Size	16-Bit
Speed	32MHz
Connectivity	I ² C, SPI, UART
Number of I/O	21
Program Memory Size	32 KB FLASH
RAM Size	8 KB

Also, **NOT a PIC16!**

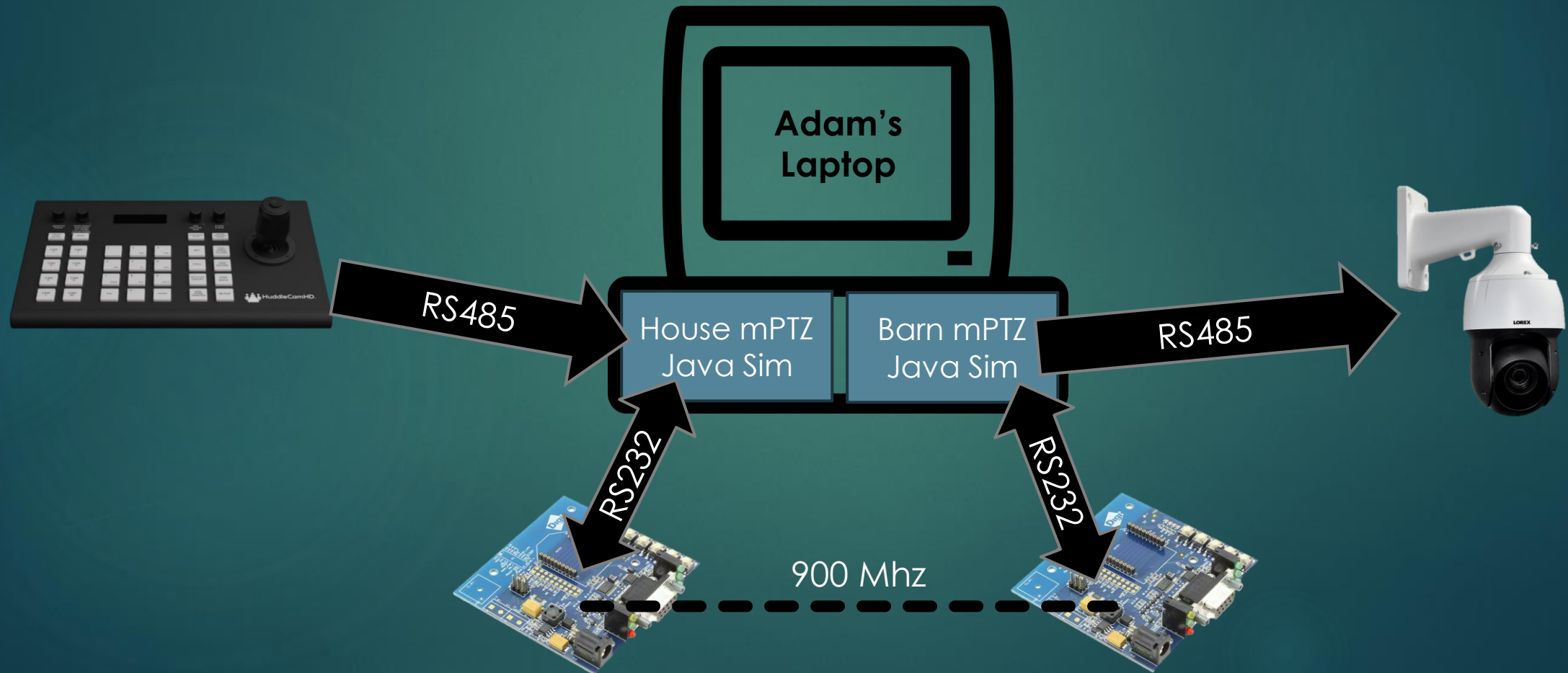
Story Time – The Product

I was contracted to create the mPTZ (multi Pan Tilt Zoom) hardware and software. It was basically a smart wireless connection between a PTZ controller and camera.



Story Time – Live Test Without PCB

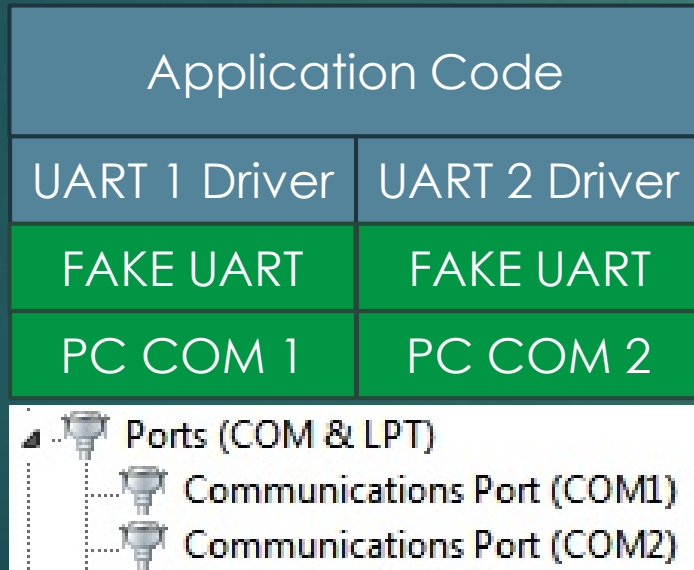
Before the custom PCB was available, Java code interfaced with real dev radio kits, PTZ controller and PTZ camera via PC com serial ports (easy with Java).



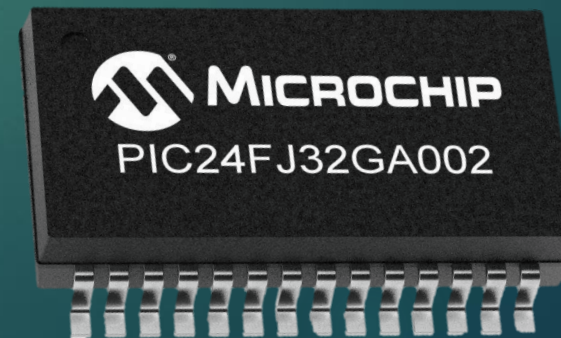
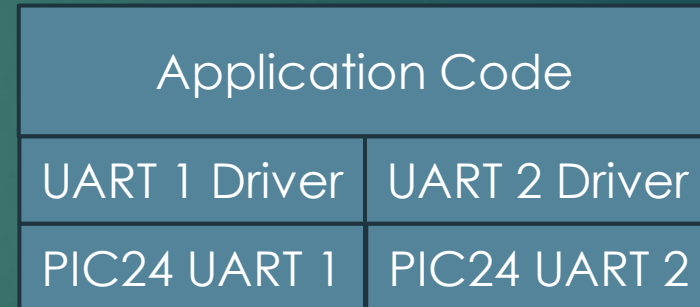
Story Time – Faking Com Interfaces

The Java code was flexible enough to use **fake** UARTs connected to PC com ports.
Generated C code used real PIC24 UARTs.

mPTZ
Java Simulation
Adam's laptop



mPTZ
C Code
mPTZ Hardware



Story Time – Simplified Data Flow

Data was sent from PTZ controller to the House mPTZ.
House mPTZ communicated (2 way) with Barn mPTZ.
Barn mPTZ sent commands to PTZ camera.



Story Time – Simulated Radio Phy

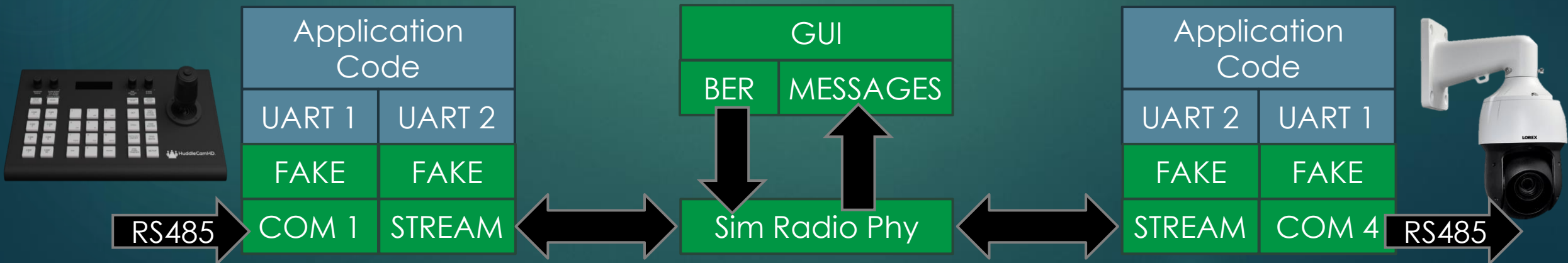
One really cool feature of this simulation setup is that I could use a simulated radio physical layer. I had a GUI where I could adjust the radio Bit Error Rate (BER) live, and see tx and rx messages live.

Full control over noise (burst, randomness, BER).

It was great for getting a **real, live feel** for how my protocol would work in challenging radio environments. Real joystick, real camera. Test responsiveness.

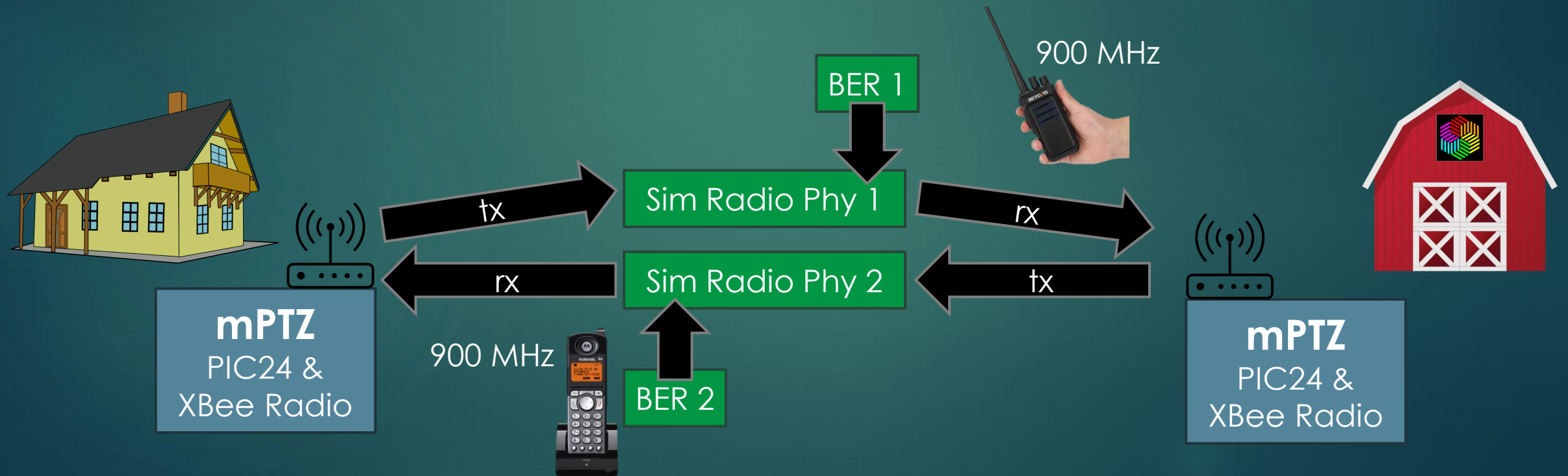
Way better than field testing. Driving miles apart, batteries, hoping for RF conditions...

Found issues and made improvements from my lab.



Story Time – Advanced Radio Phy

I made the simulation a bit more realistic to include asymmetric Bit Error Rates (BER). Even though Free Space Path Losses are symmetric, noise source locations are not. The custom protocol included acknowledgements and retries. Important to test with asymmetric conditions.



Story Time – Simulation Debugging

Each simulation ran in its own process and communicated over TCP.

A sim monitor would detect a paused simulation (hit a breakpoint) and would tell the other simulations to pause. The other simulations would resume when un paused.

Super helpful for debugging the system as a whole without one side timing out when a breakpoint was hit.

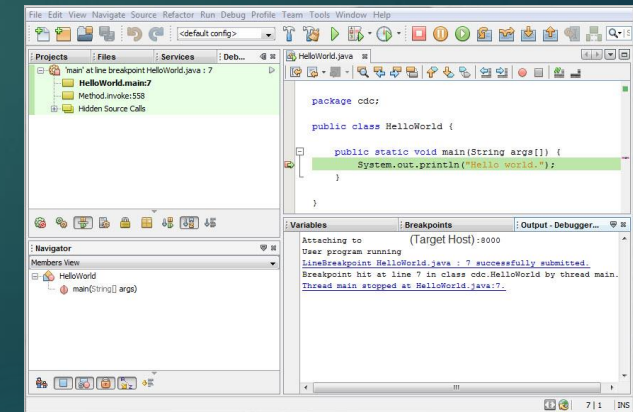
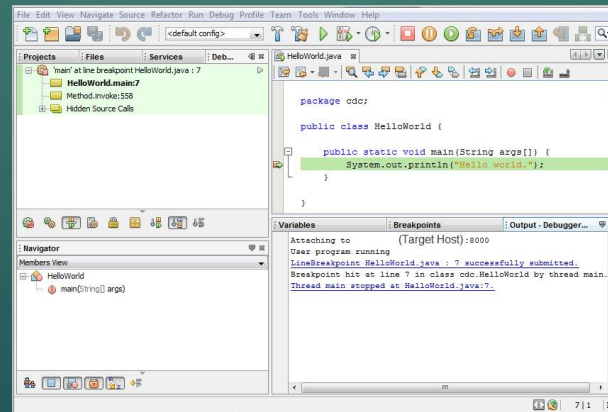
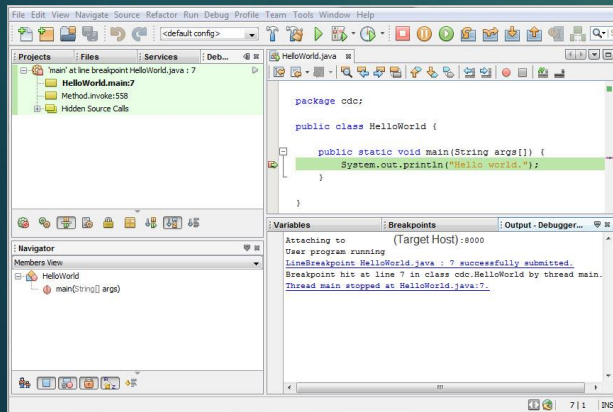
This worked, but was a bit hacky. Would like to keep all in one process.

House mPTZ Sim



RF Phy Sim & Sim Monitor

Barn mPTZ Sim



Story Time – Fuzzing Simulations

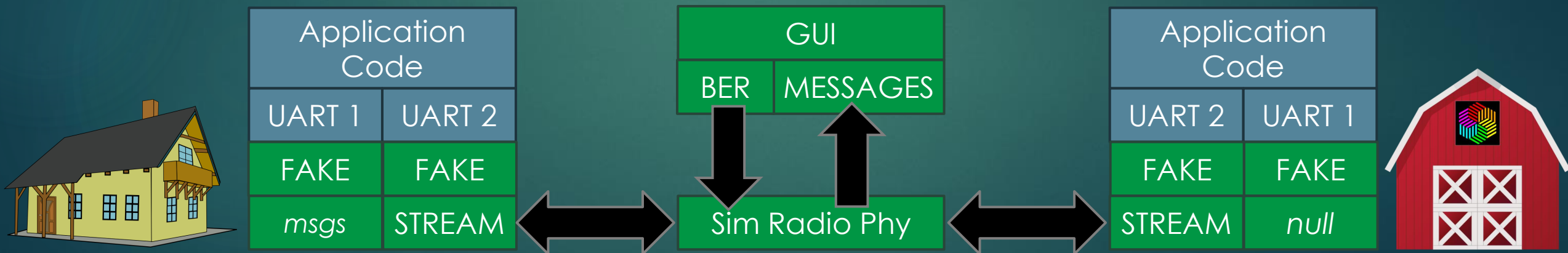
Running two working simulated controllers against each other with **fuzzing** is powerful.

Helped me find a tricky error. Set a breakpoint in error handler, then ran the simulation overnight with random injected radio errors. In the morning it had hit the breakpoint, and I could inspect the stack, variables, and message history. Easy fix once found.

Regular unit testing didn't find my mistake.

Another handy simulation feature: throwing exceptions with info and stack traces.

No exceptions in generated C code.



Story Time – C Code

The Java code actually wrote directly to PIC24 peripherals. No hand written C code.



C written as Java

```
public static void enable() {  
    PIC24FJ._U1TXIF = 0;  
    PIC24FJ._U1RXIF = 0;  
    PIC24FJ.U1MODEbits.UARTEN = 1;  
    PIC24FJ.U1STAbits.UTXEN = 1;  
}
```

Story Time – Success

The result of easy unit testing and simulations:

- ▶ No bugs found in the field
- ▶ No expensive field updates
- ▶ Happy client
- ▶ Happy farmers
- ▶ Happy me!

Why Not Java to C Then?

- ▶ Java lacks unsigned integers needed for embedded systems.
 - ▶ In the end, it worked, but I didn't see a future for it with embedded.
- ▶ C# has features that we can use for things like
 - ▶ C++ style destructors (RAII) with ``using`` statements.
 - ▶ Detecting memory issues like use after free with full stack traces.
 - ▶ Implementing custom numeric types (saturating, error on overflow...).
- ▶ C# generators should allow us to overcome problems we might face
- ▶ C# analyzers and code fixes are awesome and will also help overcome issues

Still Interested?

Video 4/6 is up next... See description for links.

Write Embedded C
Faster, Better, Safer
With Fin

AN INTRODUCTION



Fin Language
Features

SOME OF THE GOOD STUFF



Multiple Controller
Simulations FTW

JAVA TO C99 (10 YEARS AGO)



Embedded C
Software Testing
& Design Choices

WITH ASSOCIATED CHALLENGES

Language
Comparison

NO LANGUAGES WERE HURT...



Fin Implementation
And Challenges

WE CAN DO IT!





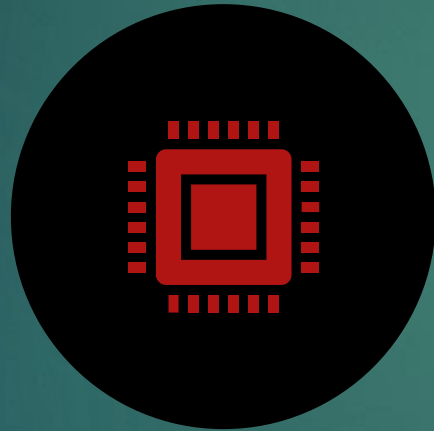
Embedded C Software Testing & Design Choices

WITH ASSOCIATED CHALLENGES

How to test our embedded code?

Two main choices.

On-Target Testing



Testing on the actual hardware & microcontroller.

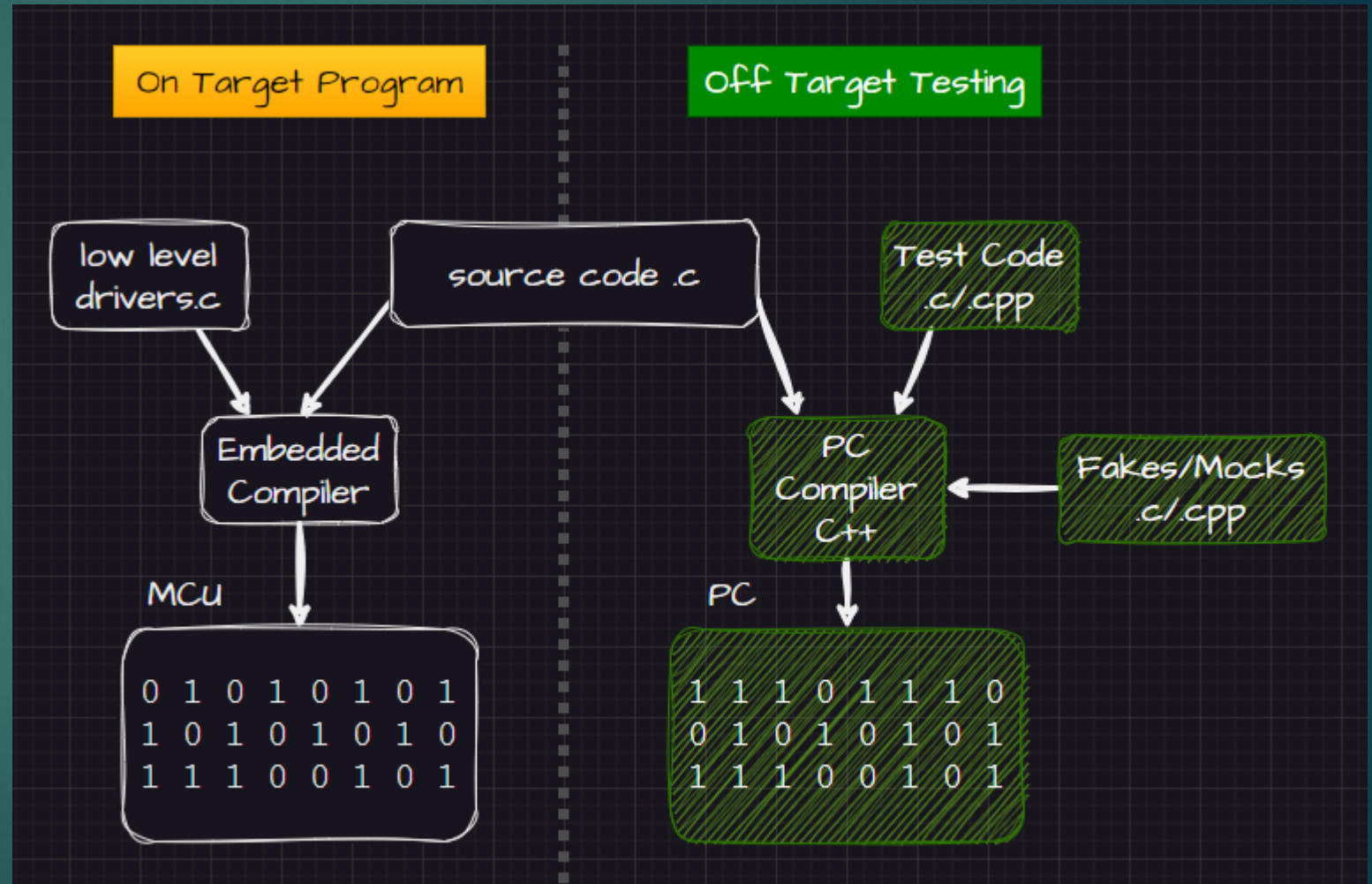
Off-Target Testing



Testing application logic on a computer.

C - Dual Target to Test Off-Target

- ▶ Off-target testing is basically a simulation
- ▶ Application logic is a prime candidate for off target testing
- ▶ Diminishing returns on simulating low level drivers



Off-Target Testing Benefits

Testing off-target has a lot of advantages

- ▶ Easy automated testing - **HUGE BENEFIT!!!**
- ▶ Faster compile and test cycles (no need to wait for flashing)
- ▶ Can develop without hardware (useful if hardware not ready)
- ▶ More powerful debugger (unlimited breakpoints)
- ▶ Supports Continuous Integration (build server can run tests on git push)

Example Product

It was a peaceful summer night. Nice breeze blowing in the windows.

My kids were soundly sleeping and I was relaxing.

My perfect C program

```
#include <stdio.h>

int main(void) {
    (void)printf("In your face U.B.!\n");
    return 0;
}
```

The Problem(s)

When a wandering hoard of drunken teenagers came by shouting obscenities and arguing over who's more drunk.



I pulled the curtains back to see what the fuss was all about, my unsupervised program promptly segfaulted and one the little turd-burglers threw a Bluetooth speaker at me (true).

In that moment, inspiration for a new product struck me!

GET-OFF-MY-LAWN-1000 (GOML)

It combines a good solid haranguing with a powerful water canon for maximal effect.

GET OFF MY LAWN

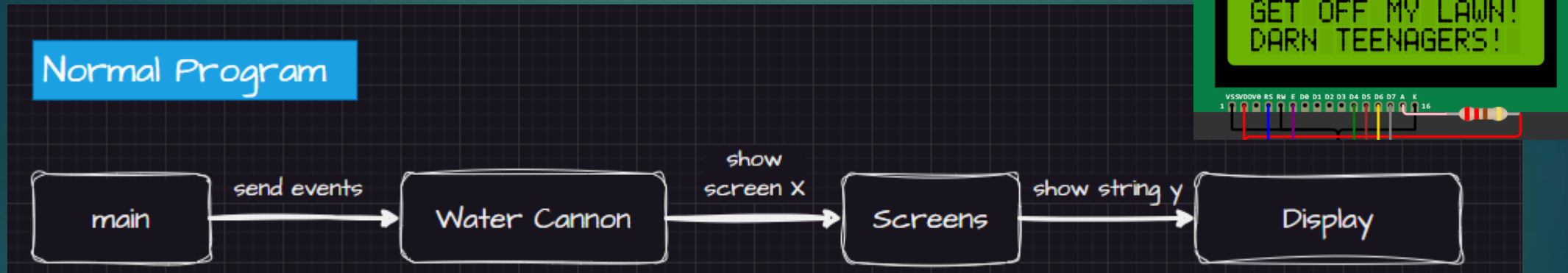


+



GET-OFF-MY-LAWN-1000 Features

Very simple example program



- ▶ detect button presses

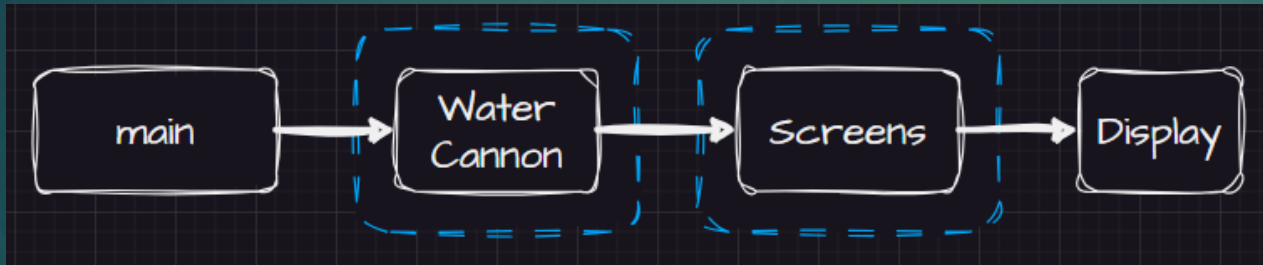
- ▶ handle events

- ▶ splash screen
- ▶ home screen
- ▶ idle screen
- ...

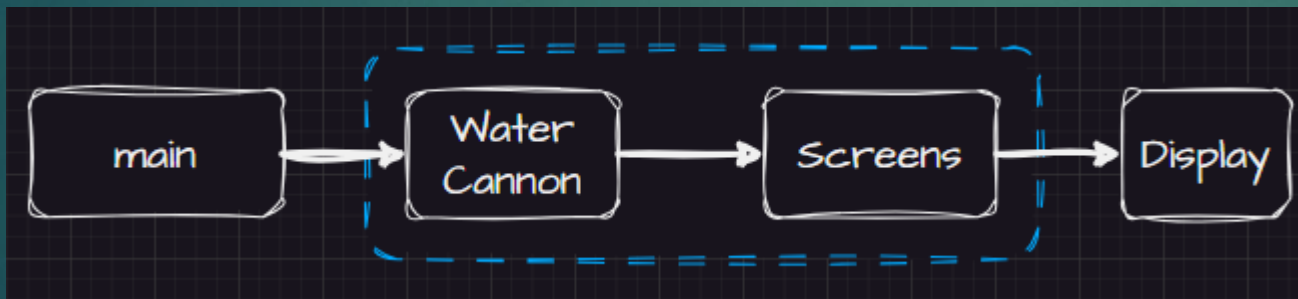
- ▶ set title
- ▶ set subtitle

Types of Software Tests

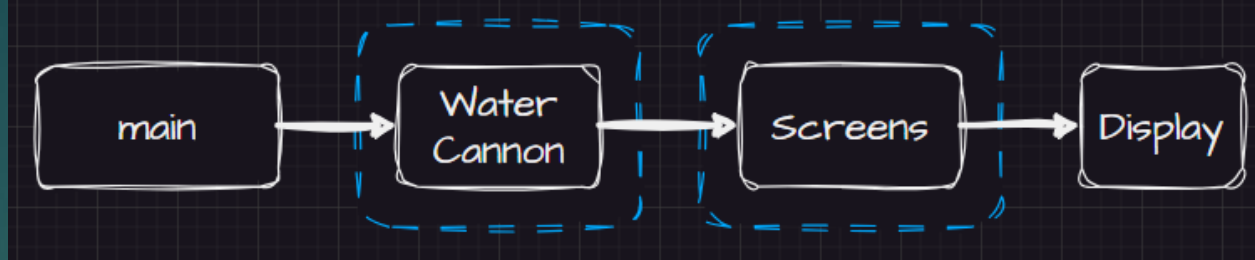
- ▶ Unit tests - Isolated module tests



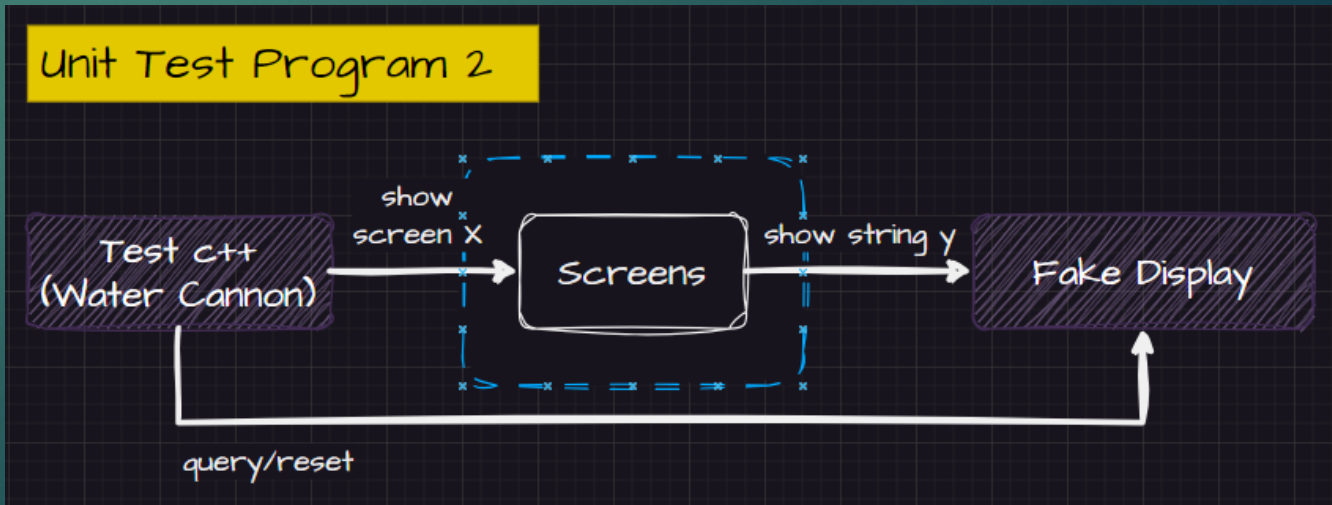
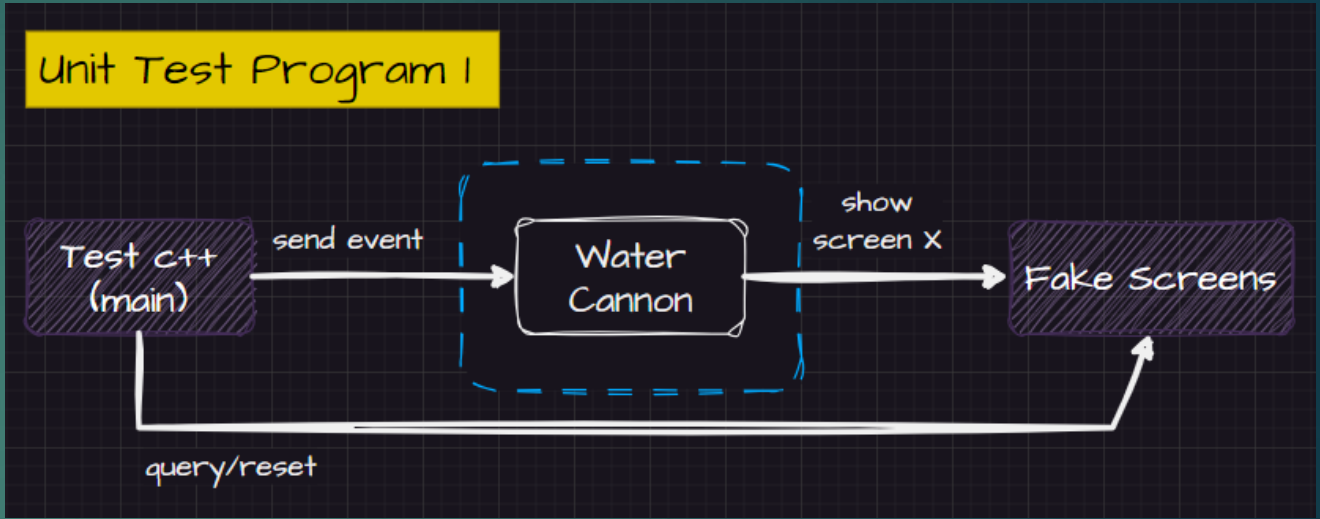
- ▶ Integration tests - Are units/modules connected properly?



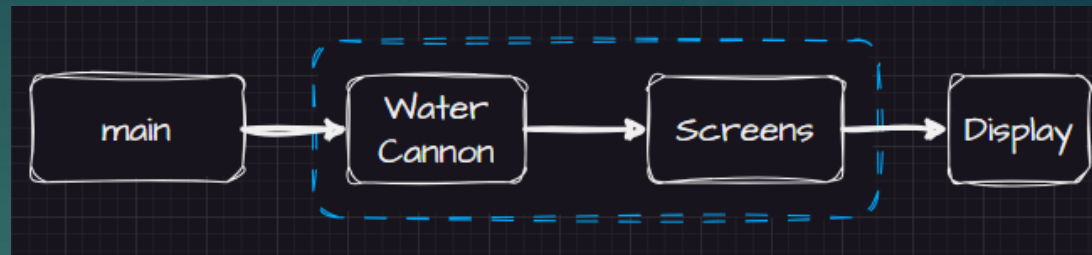
Unit Tests



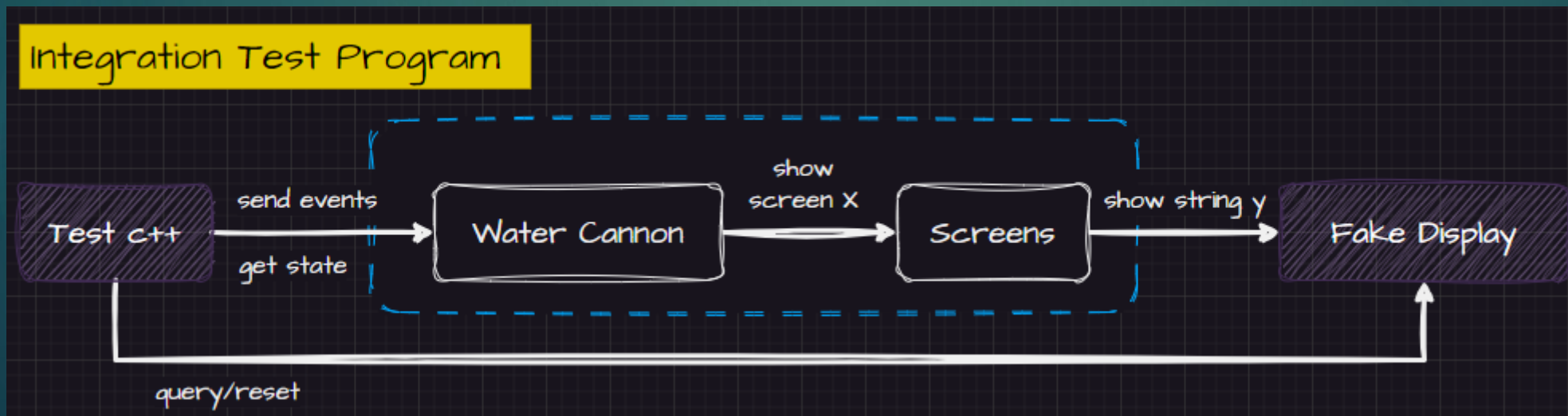
- ▶ Isolated module tests
- ▶ Each module's dependencies need to be faked/mockd
- ▶ Very useful during development
- ▶ Less sensitive to changes in other modules
- ▶ A lot of work in C due to large number of fakes/mocks required.



Integration Test



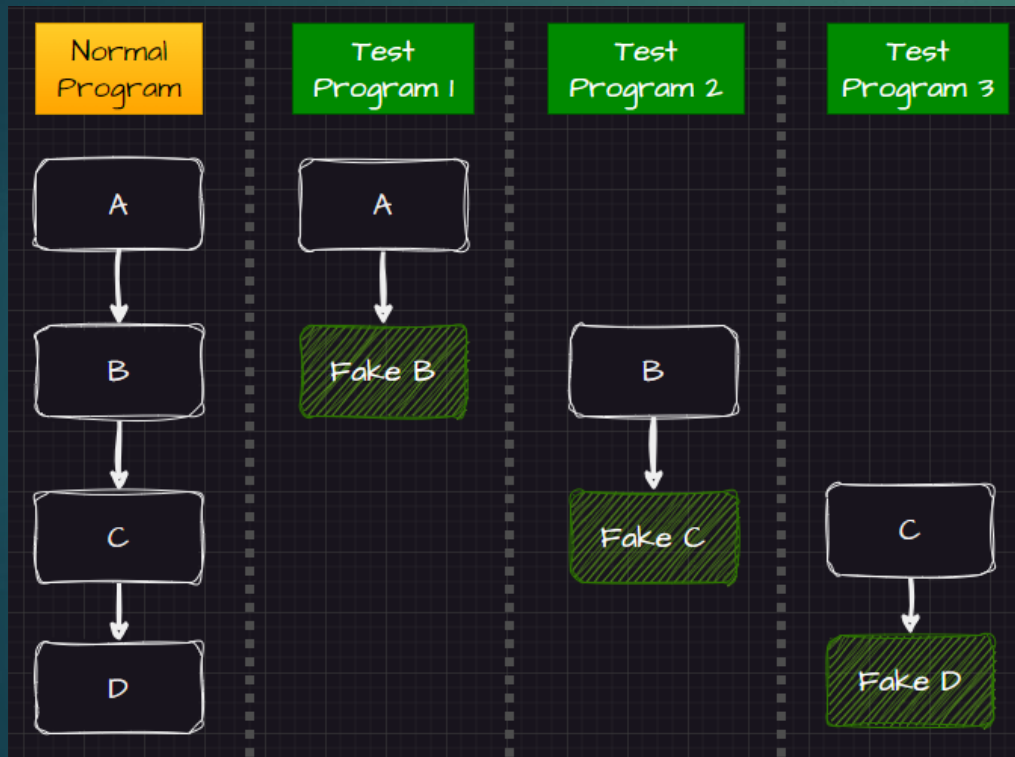
- ▶ Tests the integration of modules together (Water Cannon & Screens).
- ▶ Can be much less work than unit testing due to fewer fakes/mocks.
- ▶ Tests break often during development. A small change can break all tests. Can be mitigated with well designed C++ test fixtures.
- ▶ Royal mess if not done well. Poorly written integration tests are like chains.



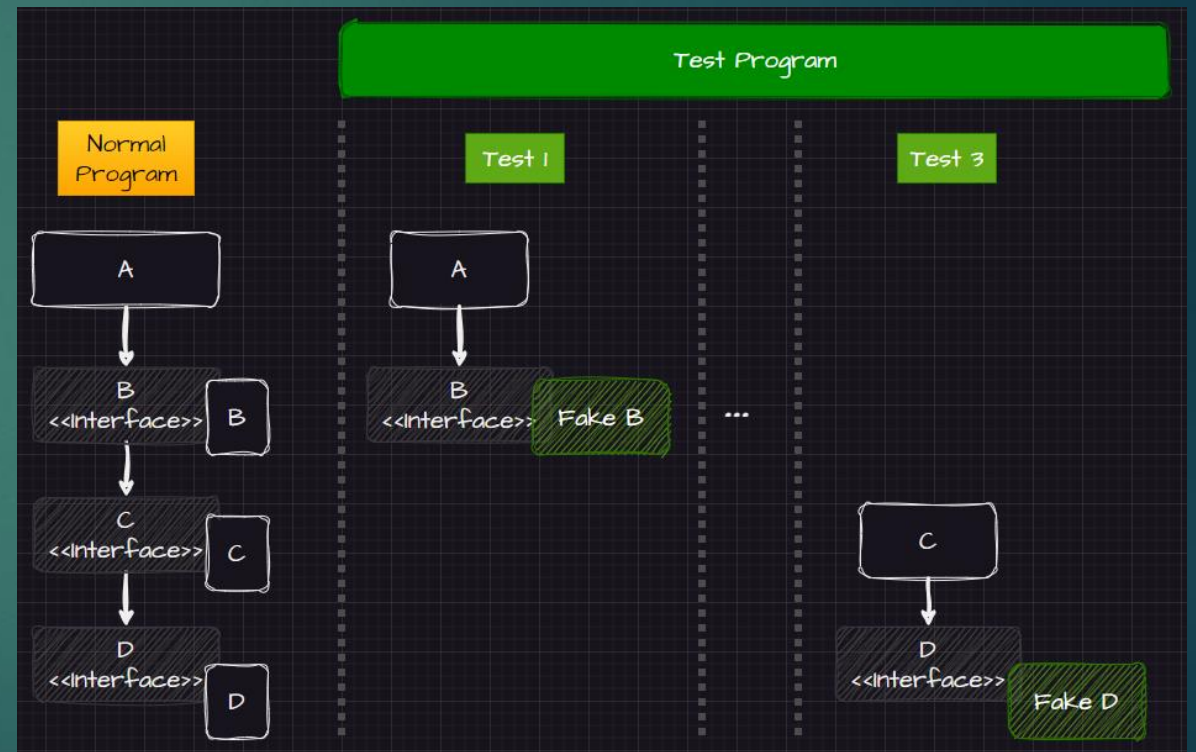
C - Faking/Mocking - Approaches

Both Integration and Unit tests require fakes/mocks.

1. Link time substitute



2. Function pointer substitute (object oriented)

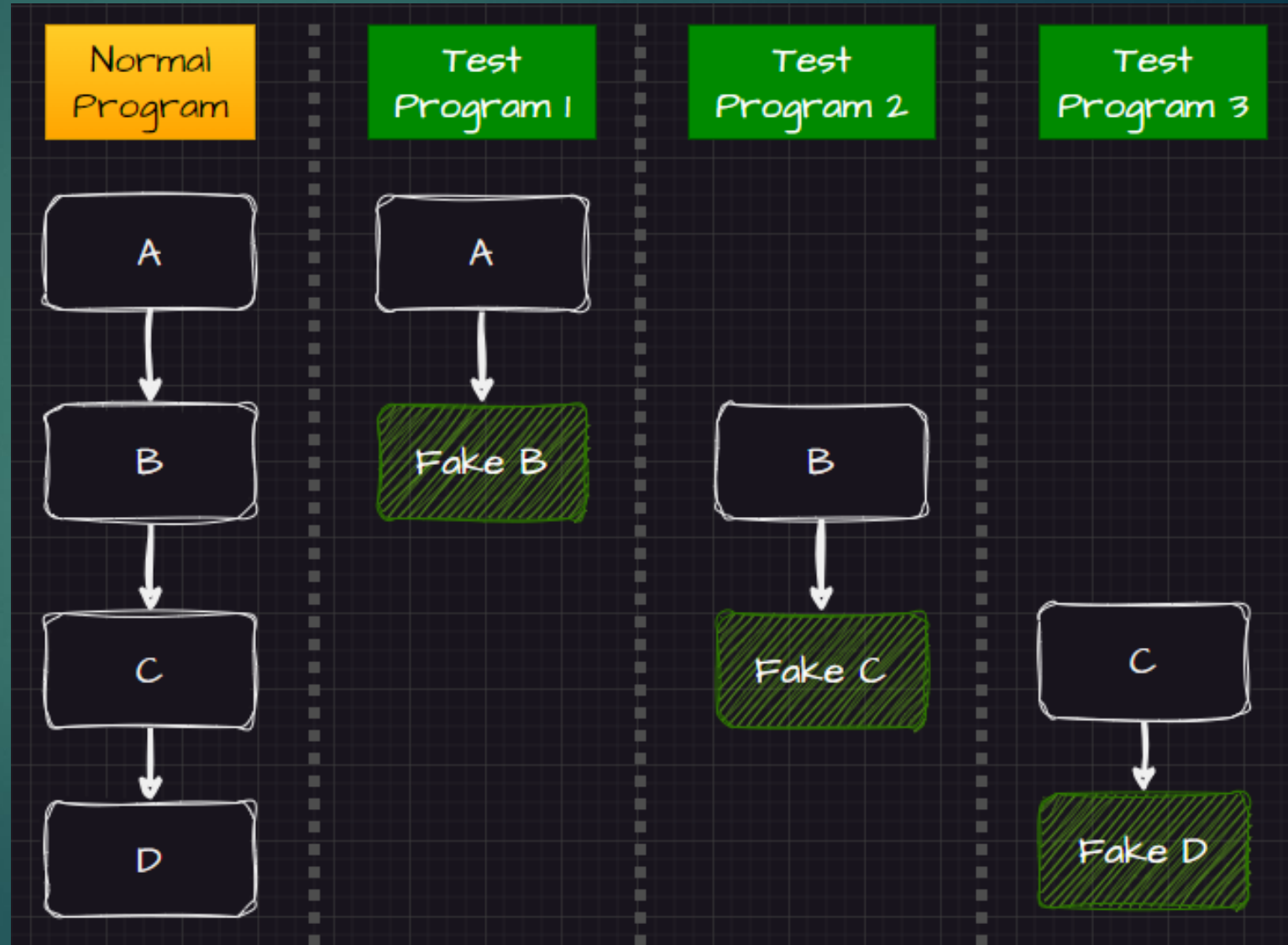


3. Last resort – preprocessor



C - Faking/Mocking – Link Time

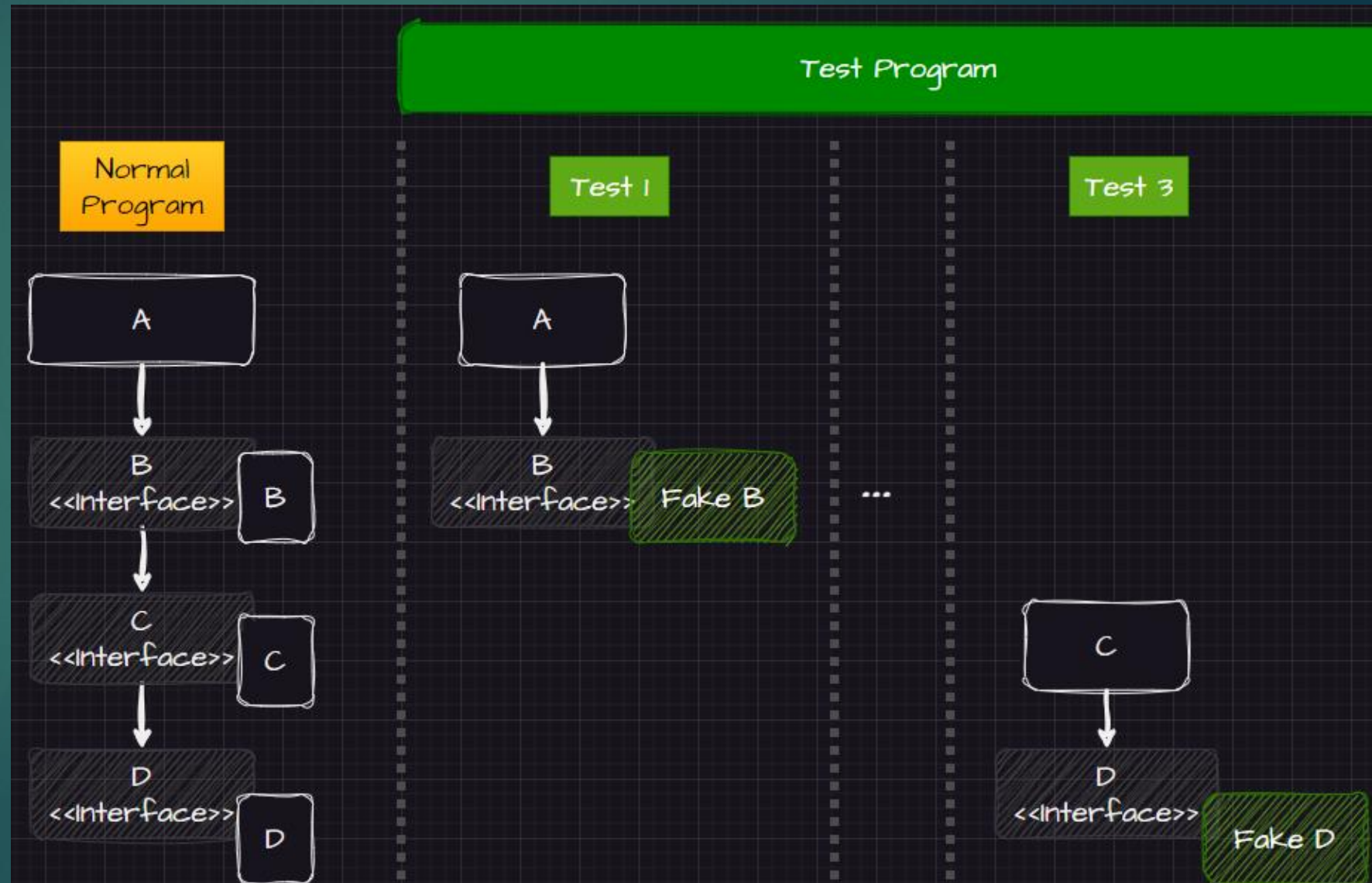
- ▶ Use linker to substitute in fake.
- ▶ Straightforward, doesn't affect how you write code.
- ▶ No code/RAM/speed cost.
- ▶ To fake different layers, we must link and run multiple test programs.
- ▶ Adds complexity and burden to test build system.
- ▶ Compiling and executing multiple programs slows down testing.
- ▶ Can't mix real B and Fake B in test/simulation.



C - Faking/Mocking – Func Pointers

- ▶ Assign fakes at runtime.
- ▶ Only a single test program is required.
- ▶ Most flexible for testing.
- ▶ Code/RAM/speed cost.
- ▶ Lots of tedious work creating/maintaining virtual tables.
- ▶ Production code more difficult to understand and walk through.

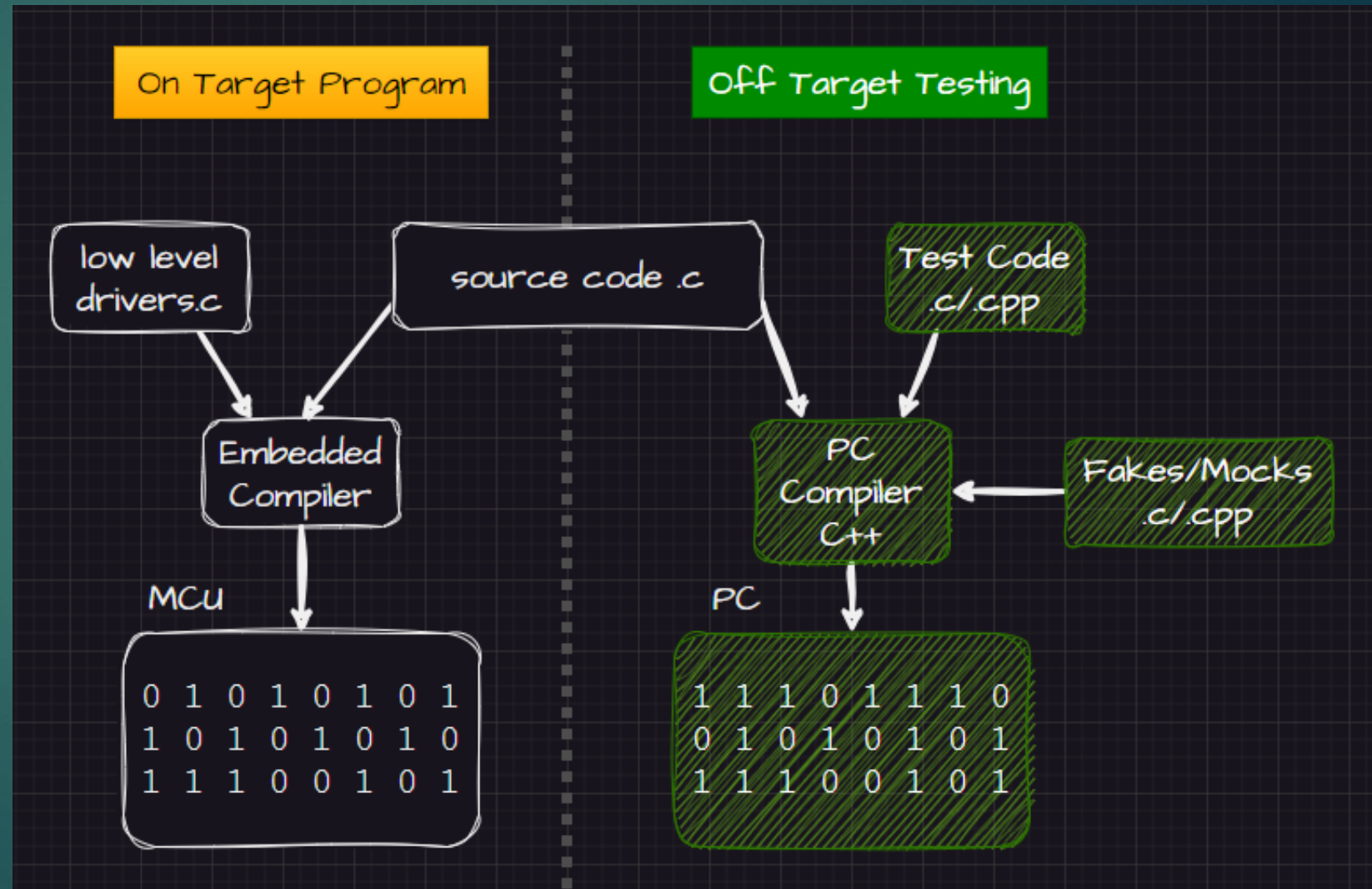
There are more function pointer approaches. This is the most capable.



C - Dual Targeting Challenges

Dual targeting does come with some challenges:

- ▶ Different int sizes
- ▶ Different enum sizes
- ▶ Different packing syntax
- ▶ Different compiler bugs/quirks
- ▶ Different runtime libs
 - ▶ String functions
- ▶ Byte ordering and bit field differences
- ▶ Undefined Behavior
- ▶ ...



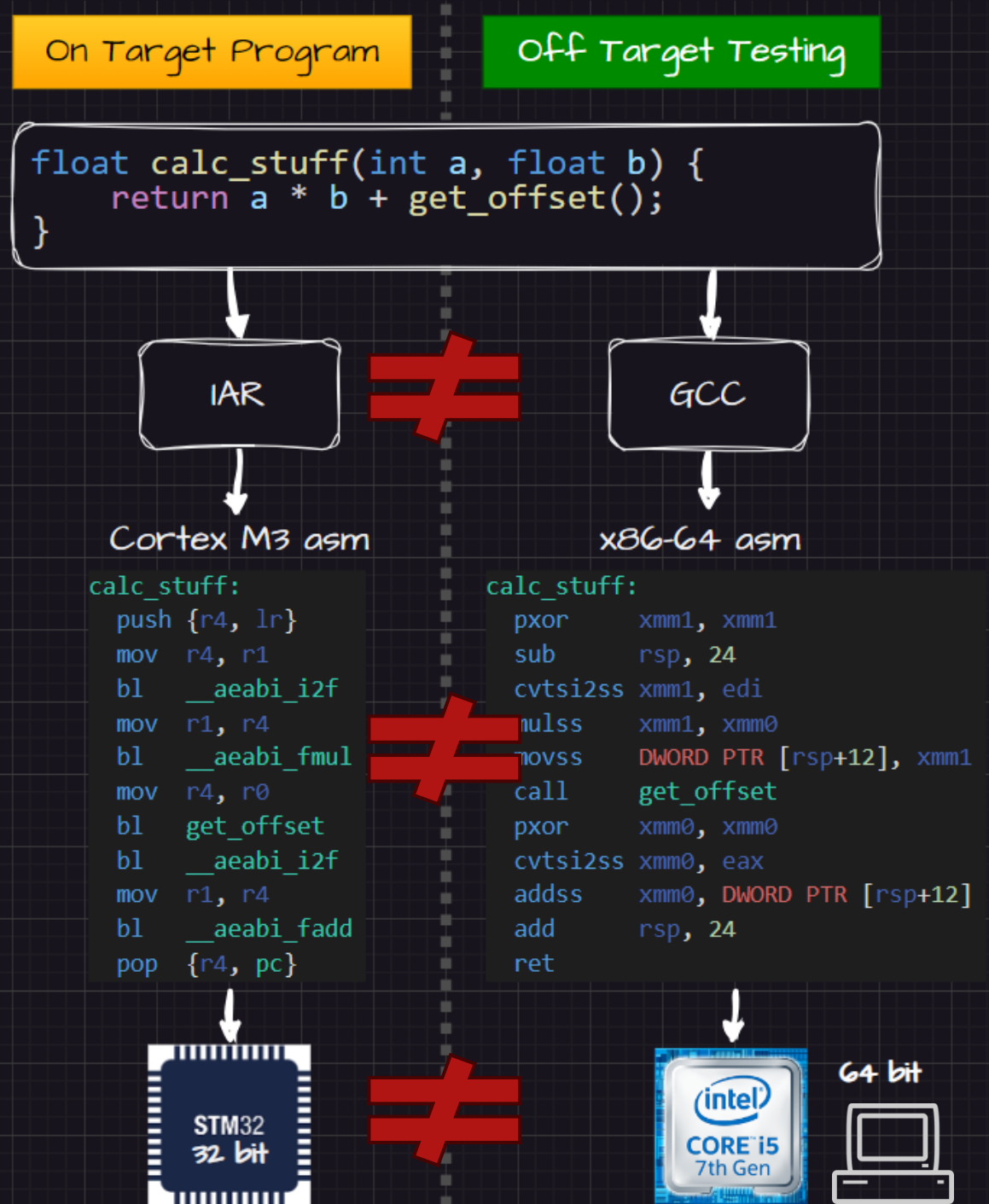
C - Test Accuracy

The compilers are different.

The assembly is different.

The processor architecture is different.

What are we really testing?



What are we really testing?

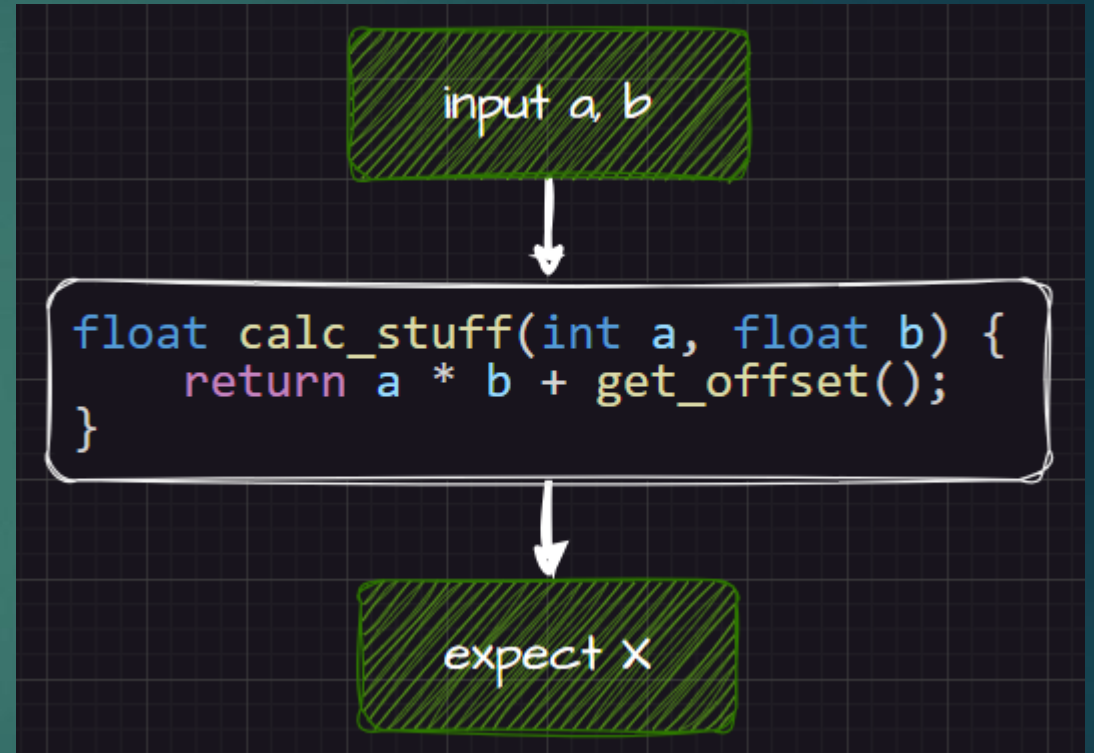
Testing the programmer's intent.

Did we make a mistake?

Our tests are essentially a simulation.

If we are already simulating, **why not use something that makes things easier?**

Something like fin :)



Testing C Code Is A Chore

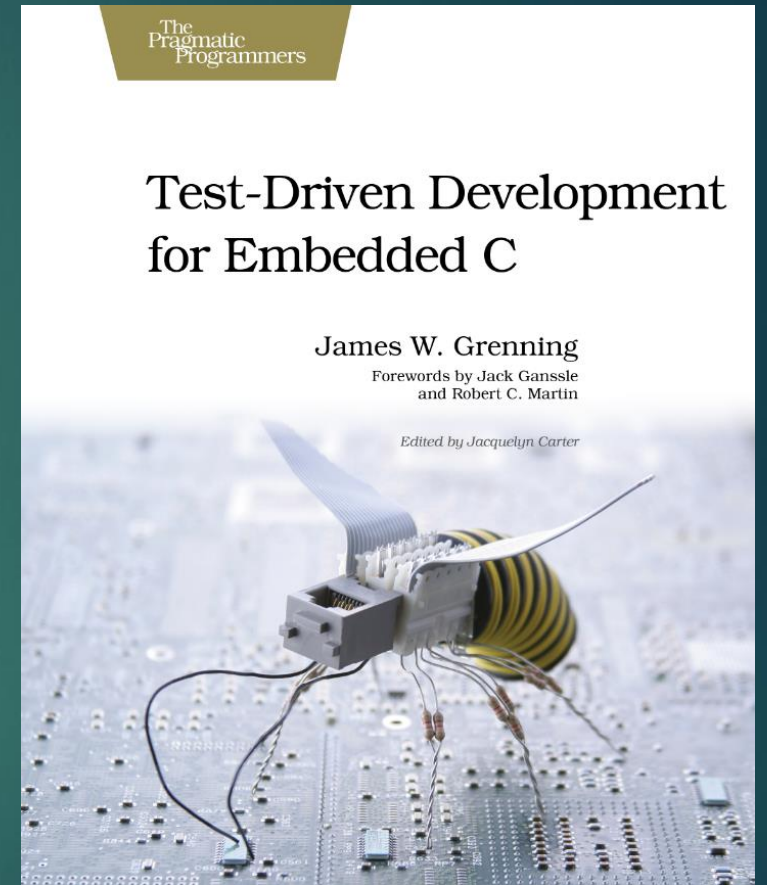
Writing C is usually fun.

Testing C is usually **NOT**.

Testing embedded C can be even more difficult as resource constraints can be tough.

Entire books are written about it. Unfortunately, even with all the knowledge in books, testing C code almost always has a negative trade off.

Testing is still worth it though.



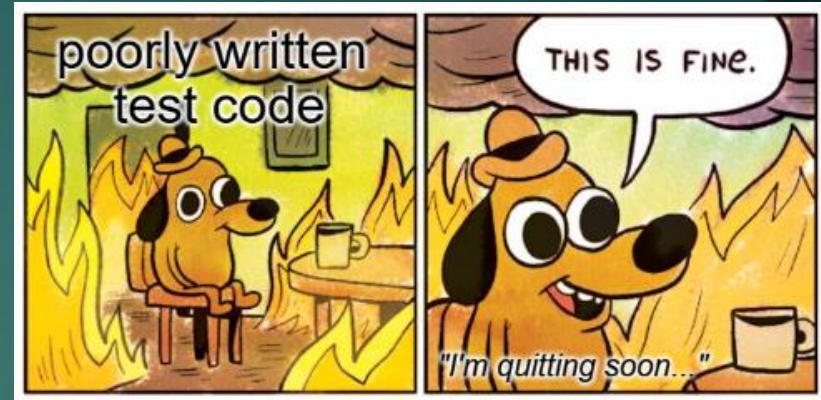
Good book!

Software Tests Are Worth The Pain

- ▶ Software tests can help save us from “*Debug-Later-Programming*” problems.
- ▶ If we don't write software tests, we commit to a lot of debugging latter.
- ▶ With Debug-Later-Programming:
 - ▶ Feedback revealing mistakes may take days, weeks, or months to get back to us.
 - ▶ Testing on embedded hardware is slow so we release numerous changes together.
 - ▶ The more changes in a build, the harder a mistake (or mistakes) can be to find.
 - ▶ James Grenning TDD

Tests Code Matters

- ▶ When people are first getting started writing software tests, they often neglect them.
- ▶ *“It’s just test code. It can smell like garbage and violate good programming practices.”*
- ▶ Once we write enough tests, this approach punishes us.
- ▶ Want to add a new simple feature? Add 4 hours for inflexible tests.
- ▶ Want to change a sensor? Add 8 hours for updating super redundant tests.
- ▶ Want to change something fundamental? Add 16 hours for test rework.



Sick of poorly written tests holding you and the product back?

- ▶ Strong tooling/refactoring can help a lot (if supported by the language)

Or Just Delete Them!

- ▶ Kidding not kidding... You never commented out a failing and terribly written test?
 - ▶ Add ticket, explain the problem.
 - ▶ Meet the project deadline, move on.
 - ▶ Maybe one day have time to fix.
- ▶ Refactoring poorly written C code is slow.

Why do we write crappy tests?

Because writing good tests is hard.

Writing good tests in C is harder.

Writing good tests in fin is much easier.



BAD TEST CODE

Fin - Testing Improved

As shown in video 1. Super simple with auto mocks. No hassle.

fin code (C#)

```
public class Screens
{
    Display _display;

    public Screens(Display display)
    {
        _display = display;
    }

    public void show_idle()
    {
        _display.set_title("Idle");
        _display.set_subtitle("Press OK");
    }
}
```

Test/simulation code (C#)

```
public class ScreensTest
{
    [Fact]
    public void ShowIdle()
    {
        // arrange
        var mock_display = Substitute.For<Display>();
        Screens screens = new Screens(mock_display);

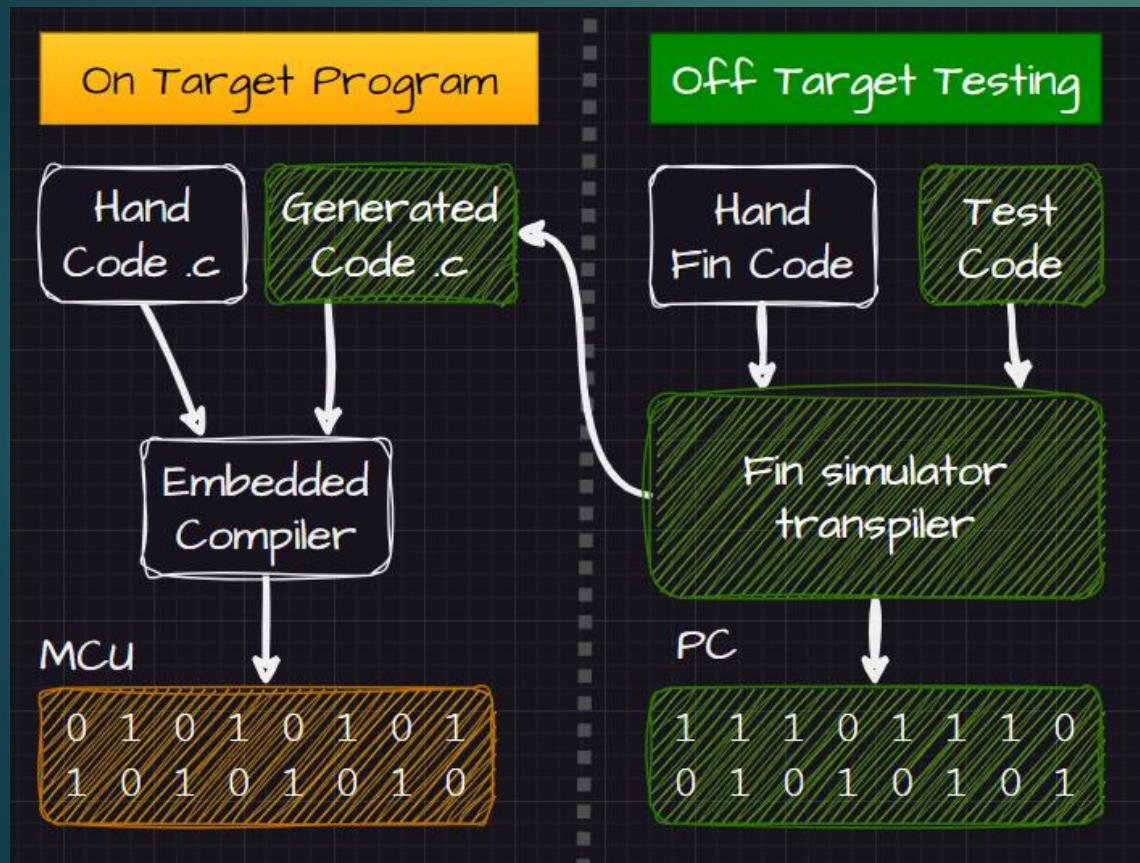
        // act
        screens.show_idle();

        // assert expected calls received by mock
        mock_display.Received().set_title("Idle");
        mock_display.Received().set_subtitle("Press OK");
    }
}
```

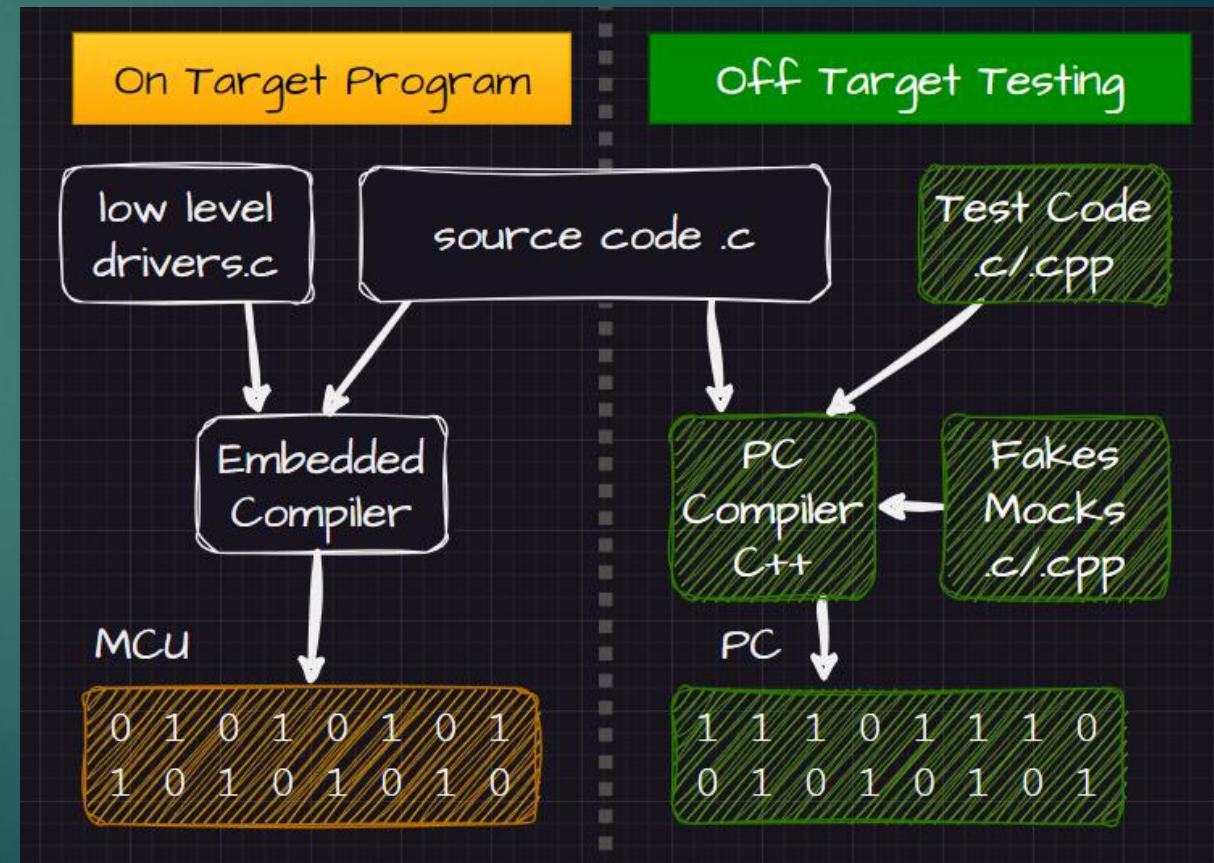
Fin workflow

Fin simulation/testing is **fast!** Do that often, test on hardware when needed.
Similar workflow to conventional dual targeting for testing.

Improved Testing With Fin



Conventional Dual Target For Testing C/C++



C - Single or Multi Instance?

▶ Single Instance

- ▶ Use when you only ever want a single instance of a module.
- ▶ Variables can be allocated at file scope.
- ▶ Functions are simple. `bool WaterCannon_is_calibrated(void)`

▶ Multi Instance

- ▶ Use when you need or may need multiple instances of a module.
- ▶ Variables are kept in a struct.
- ▶ That struct is passed to all functions (speed, stack usage).

```
bool WaterCannon_is_calibrated(struct WaterCannon *self)
```

C - Single or Multi-Instance?

- ▶ Single Instance is generally recommended until Multi is needed.
- ▶ Refactoring from single to multi can be a pain in large code bases.
 - ▶ All variables need to be moved to a struct.
 - ▶ All uses (internal/external) need to be updated.
 - ▶ Test code will need to be updated.
 - ▶ Doable, but can be lots of work still.



With `fin` we write modular multi-instance code, and `fin` optimizes if only one instance is needed.



Highly Flexible/Reusable Code

If we want our code to be easy to reuse, extend/adapt, then we need:

- ▶ Multi-Instance Modules
- ▶ Object Oriented Style (vtables)

There is a lot of benefit here, but what are the costs?

C - Flexibility Costs...

What we want to write...

```
static int ammo = 0;

bool WaterCannon_fire(int target) {
    // do stuff
    return true;
}

bool WaterCannon_reload(void) {
    // do stuff
    return true;
}
```

What we have
to write

```
typedef struct WaterCannon2Vtable {
    bool (*fire)(void *self, int target);
    bool (*reload)(void *self);
} WaterCannon2Vtable;

typedef struct WaterCannon2 {
    const WaterCannon2Vtable *vtable;
    int ammo;
} WaterCannon2;

const WaterCannon2Vtable waterCannon2Vtable = {
    .fire = WaterCannon2_fire,
    .reload = WaterCannon2_reload
};

void WaterCannon2_ctor(WaterCannon2 *self) {
    self->vtable = &waterCannon2Vtable;
    self->ammo = 0;
}

bool WaterCannon2_fire(WaterCannon2 *self, int target) {
    /* do stuff */ return true;
}

bool WaterCannon2_reload(WaterCannon2 *self) {
    /* do stuff */ return true;
}

// some other file
bool use_cannon(WaterCannon2 *cannon) {
    return cannon->vtable->fire(cannon, 1);
}
```

C - Writing vtables is...

...fun the first few times...

Then it starts to drag:

- ▶ Way more code
- ▶ Cognitive load for just wiring it up
- ▶ Performance costs
- ▶ Extremely verbose

```
typedef struct WaterCannon2Vtable {
    bool (*fire)(void *self, int target);
    bool (*reload)(void *self);
} WaterCannon2Vtable;

typedef struct WaterCannon2 {
    const WaterCannon2Vtable *vtable;
    int ammo;
} WaterCannon2;

const WaterCannon2Vtable waterCannon2Vtable = {
    .fire = WaterCannon2_fire,
    .reload = WaterCannon2_reload
};

void WaterCannon2_ctor(WaterCannon2 *self) {
    self->vtable = &waterCannon2Vtable;
    self->ammo = 0;
}

bool WaterCannon2_fire(WaterCannon2 *self, int target) {
    /* do stuff */ return true;
}

bool WaterCannon2_reload(WaterCannon2 *self) {
    /* do stuff */ return true;
}

// some other file
bool use_cannon(WaterCannon2 *cannon) {
    return cannon->vtable->fire(cannon, 1); // YUCK!
}
```



Zero RAM cost vtables

If your architecture supports storing data in FLASH, you can get runtime polymorphism with no additional RAM cost. Very cool!

The downsides is that the C code shouldn't be hand written.

- ▶ vtables have to be global and connected by user code (not in abstraction)
- ▶ Even more verbose
- ▶ High chance of mistake
- ▶ Good if used with code generation (fin)

Still Interested?

Video 5/6 is up next... See description for links.

Write Embedded C
Faster, Better, Safer
With Fin

AN INTRODUCTION



Fin Language
Features

SOME OF THE GOOD STUFF



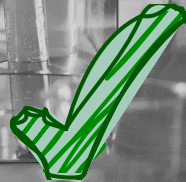
Multiple Controller
Simulations FTW

JAVA TO C99 (10 YEARS AGO)



Embedded C
Software Testing
& Design Choices

WITH ASSOCIATED CHALLENGES



Language
Comparison

NO LANGUAGES WERE HURT...



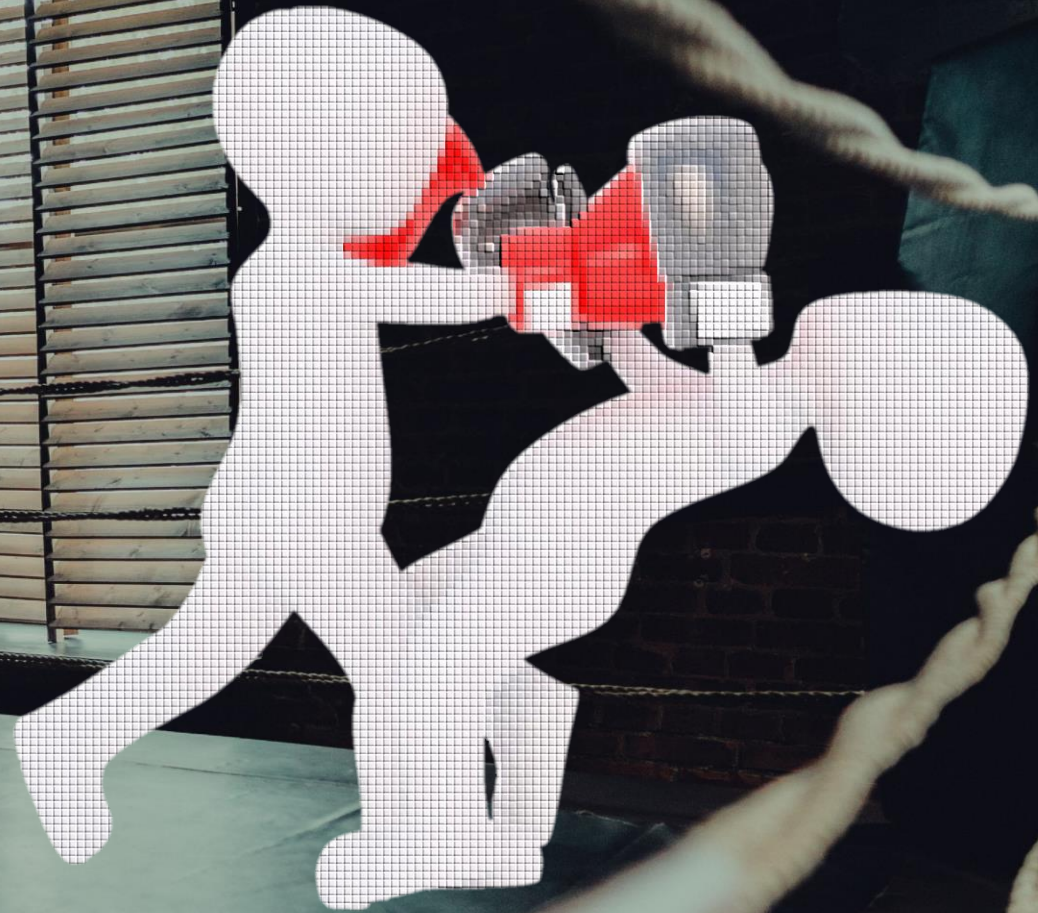
Fin Implementation
And Challenges

WE CAN DO IT!



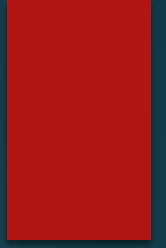
Language Comparison

NO LANGUAGES WERE HURT...



Experiences May Differ

This is based on my experience and research.



Why not use C++?

C++ is incredibly powerful in highly skilled/trained hands.

C++ can be incredibly **destructive** to projects with inexperienced teams.



There is no shortage of C++ criticisms. Horrific compiler errors. Poor refactoring...

Poor C++ Toolability

Don't just take my word on it.

Herb Sutter (a prominent C++ expert) is working on a new c++ variant **cppfront**

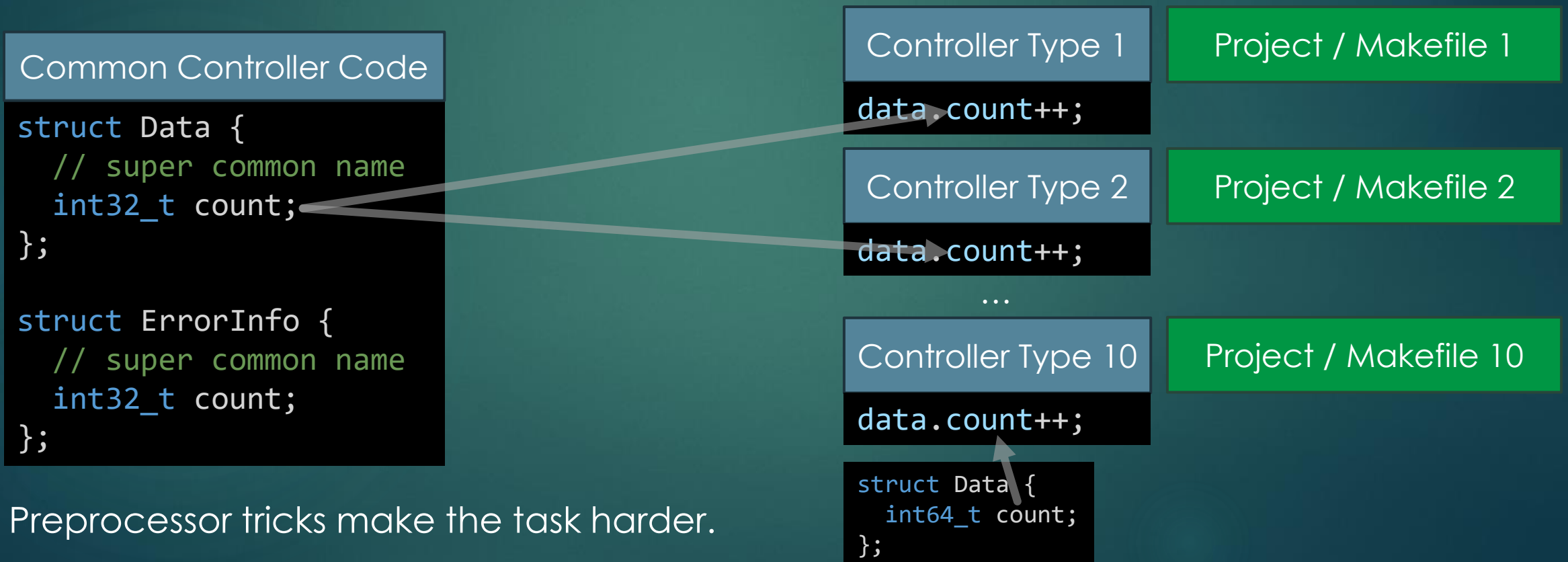
- ▶ 10x simpler, 50x safer, 10x toolability. This is a worthy goal.
- ▶ Glad Herb is working on this. Definitely needed.
- ▶ Experimental, may be years before usable

Toolability - Multiple Project Refactoring

Mono-repo with multiple projects that share common code.

Modern tools often fail to refactor elements with a common name.

What should be a simple refactor rename can introduce bugs. It should just work.



Preprocessor tricks make the task harder.

Simple Refactoring C/C++ Is Hard

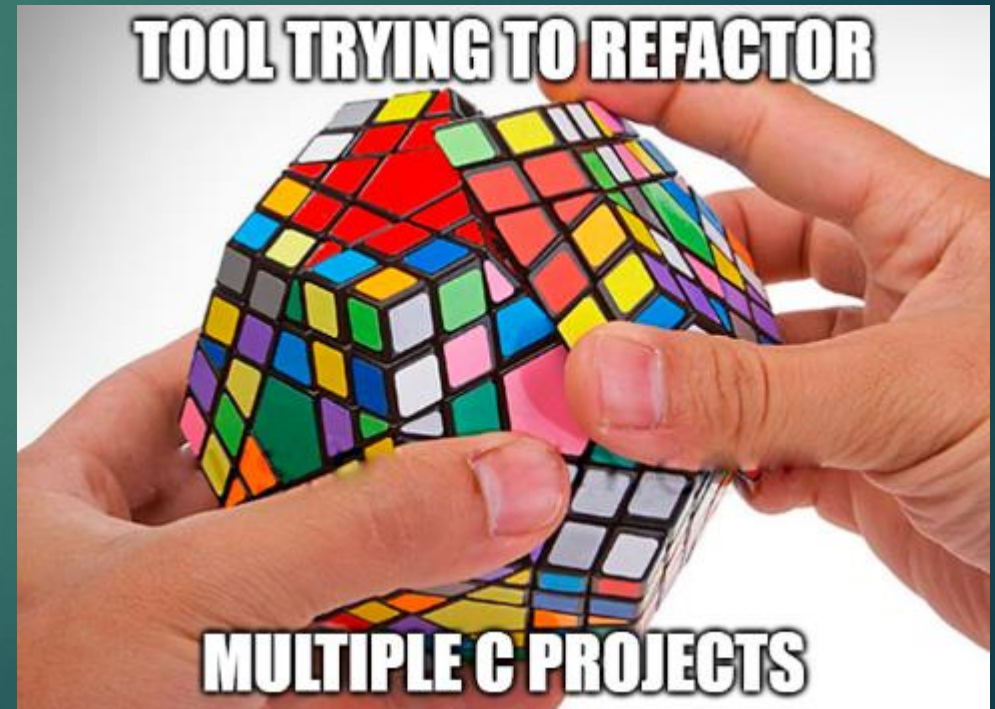
This is a non-trivial problem for tool authors.

In some cases, impossible. Inherent with C/C++.

Try to rename a field or function and the IDE brings up a large list of potential matches.

Fin gives us fearless refactoring.

- ▶ Fully accurate.
- ▶ It takes a few seconds max.



The Best C++ IDE?

How does the best C++ IDE in the world solve this problem?

It hasn't yet...

CPP-1537 Created by Alexey Ushakov 9 years ago Updated by Zendesk Integration 2 weeks ago

Support multiple projects

Relates to 11 Is required for 1 Depends on 2 Is duplicated by 4 Parent for 5

RELATES TO 11 ISSUES (7 UNRESOLVED)

- CPP-7438 Support for catkin
- CPP-8414 Specify scripts to use for CMake generation (instead of directly invoking cmake)
- CPP-10513 Ctrl+B variation for jumping to project where type is defined
- CPP-11524 Generate inclusion guard name depending on the selected target/project
- CPP-12785 Allow adding several content/sources roots



CLion

Do you mean to have an ability to perform IDE actions (lookup, navigate, refactor ...) on several projects at the same time?

Yes, mainly because I usually develop both the libraries and the apps. This is a really needed feature for at least the embedded devices world =).

Tools Overrated - Refactoring for wimps?

Real programmers write it right the first time, right?

- ▶ Refactoring is key to TDD (nice idea)
 - ▶ red, green, refactor cycle
- ▶ Software should be “soft” – easy to change
- ▶ Improving code is necessary to live a good life
- ▶ Better refactoring => better life



**Refactoring
Code**



**Get code
working.
Never ever
touch it again.**

What about other languages?

Very few languages were designed with embedded systems in mind.

Most modern languages are way too heavy and/or require a garbage collector.

There are a few (rust, zig) that target LLVM, but many embedded targets not supported.

We need to compile to C for embedded systems to ensure portability.

Some can compile to C, but it isn't easy to understand. "Fun" debugging on target.



Fin generates high quality C99 code that follows your input as much as possible.

Fin - Benefits of Compiling to C

I actually don't see compiling to C as a bad thing. It is good for many reasons.

Some of the most powerful parts of languages like C++ and rust feel like **FREAKING MAGIC!**

It can be difficult to understand the costs of using some of those features.



With fin, you can look at the highly readable C. You don't have to delve into assembly. You can easily audit and understand EXACTLY what is happening.

You can also use all the existing analysis and tools developed for C.

Still Interested?

Video 6/6 is up next... See description for links.

Write Embedded C
Faster, Better, Safer
With Fin

AN INTRODUCTION



Fin Language
Features

SOME OF THE GOOD STUFF



Multiple Controller
Simulations FTW

JAVA TO C99 (10 YEARS AGO)



Embedded C
Software Testing
& Design Choices

WITH ASSOCIATED CHALLENGES



Language
Comparison

NO LANGUAGES WERE HURT...



Fin Implementation
And Challenges

WE CAN DO IT!



The background of the slide features a teal gradient. On the left and right sides, there are black silhouettes of two individuals rappelling down a rope. The person on the left is in a more compact, tucked position, while the person on the right is in a more extended, dynamic pose. The central text is overlaid on this background.

Fin Implementation And Challenges

WE CAN DO IT!

Won't We Lose XYZ With C#?

Because fin compiles to C, we can keep most of what we love about C

- ▶ Direct low level hardware access
- ▶ Full control over memory (no heap/GC required)
- ▶ Speed

Wait! How is this possible?

The heap is required for most C# objects.

How does fin support C like stack allocations?

fin code (C#)

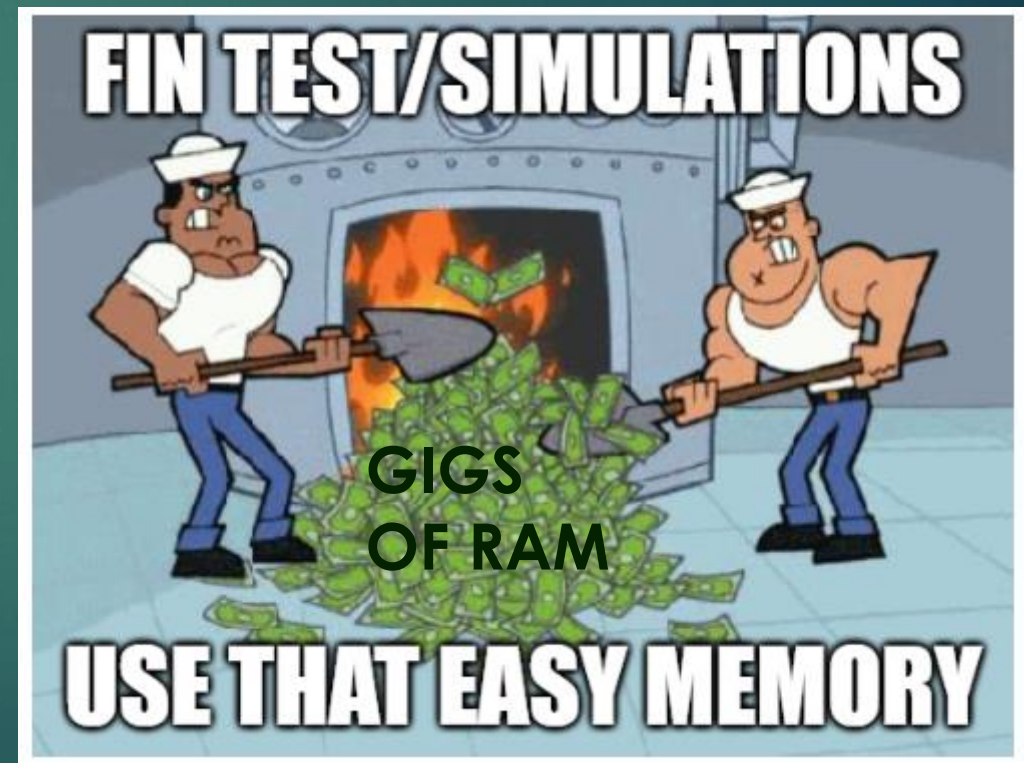
```
public void send_order(int burrito_count)
{
    using OrderPacket order = mem.place<OrderPacket>();
    order.burrito_count = burrito_count;
    order.send();
}
```

Generated C

```
void send_order(int burrito_count)
{
    OrderPacket order;
    OrderPacket_ctor(&order);
    order.burrito_count = burrito_count;
    OrderPacket_send(order);
}
```


So no heap?

To be clear, fin C# **simulations** do use the heap, but generated code does not. Having access to a garbage collector is really handy for simulations. Our test code can be developer-time efficient and rely on a GC.



`Using` Syntax?

So what's up with `using OrderPacket order?`

The C# ``using`` statement calls ``Dispose()`` on ``order`` when it goes out of scope. This allows us to mark it as garbage and detect bad memory accesses during simulation.

No memory checks in generated code (speed speed speed). 

Challenge - what about const?

C# does not have a true equivalent of C's const modifier, but we can add that. Would like to just write `const Gpio gpio`.

Const objects are important in embedded systems for MCUs that support storing configuration data in FLASH instead of RAM.

Lots of options, but I haven't settled on one yet. Feedback/suggestions would be nice.

Using custom readonly [ro] attribute

```
public void use_gpio([ro]Gpio gpio) {  
    GPIO_SetBits(gpio.port, gpio.pin);  
}
```

Using C# source generators to automatically create readonly version

```
public void use_gpio(ro_Gpio gpio) {  
    GPIO_SetBits(gpio.port, gpio.pin);  
}
```

Challenge - what about volatile?

C# does not have a true equivalent of C's volatile modifier, but we can add that.

Lots of options, but I haven't settled on one yet. Feedback/suggestions would be nice.

Using volatile `vol<T>` wrapper

```
public void use_data(vol<Data> volatile_data)
{
    using Data copy = volatile_data.load();
    do_stuff(copy);
}
```

```
public void update_data(vol<Data> volatile_data)
{
    volatile_data.store(some_copy);
}
```

Depending on data being wrapped, may require telling fin how to enter a critical section.

Could also allow direct access to data in a critical section.

Look to C/C++11 atomics.

What about XYZ?

I believe there are solutions to challenges.

Time to really kick the tires and see how it goes.

Lots of options. Feedback/suggestions would be nice.

Downside to Basing on C#

We get a huge amount of benefit from basing on C#, but there are cons too:

- ▶ Sometimes our syntax is a bit more verbose than C's
- ▶ Not bad when you consider C needs to call the constructor too.

fin code (C#)

```
public void send_order(int burrito_count)
{
    using OrderPacket order = mem.place<OrderPacket>();
    //...
}
```

Generated C

```
void send_order(int burrito_count)
{
    OrderPacket order;
    OrderPacket_ctor(&order);
    //...
}
```

Limiting Verbosity

We can improve that a bit using C# `var`. The type is inferred and easy to see.

```
using var order = mem.place<OrderPacket>();
```

We can also provide tooling so that you can just type `OrderPacket order;` and have the IDE auto complete the rest. C# makes it easy to write custom code analyzer and fixes.

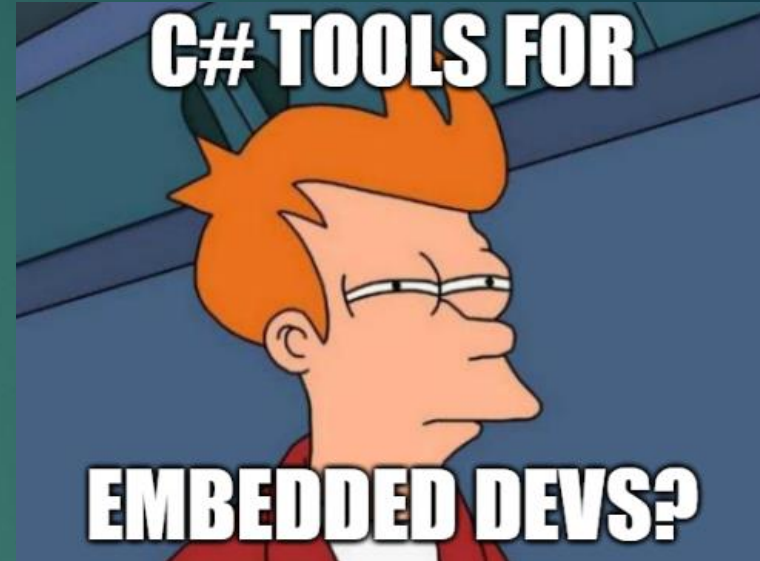
The C# compiler is open source. There are tutorials showing how to fork it and add custom syntax features that we could potentially consider in the future.

Custom Code Fixes

“Sure sure. C# toolability is great for C# devs, but not us. Users can’t extend it, right?”

Check the Roslynator open source project!

- ▶ 325 Analyzers that find issues
- ▶ 211 Refactoring suggestions
- ▶ 169 Fixes
- ▶ Community (non-microsoft) project
- ▶ Super easy to add to project (a few clicks)
- ▶ Works with vscode, Visual Studio, Rider



Roslynator is a set of code analysis tools for C#, powered by Roslyn.

josefpihrt.github.io/docs/roslynator

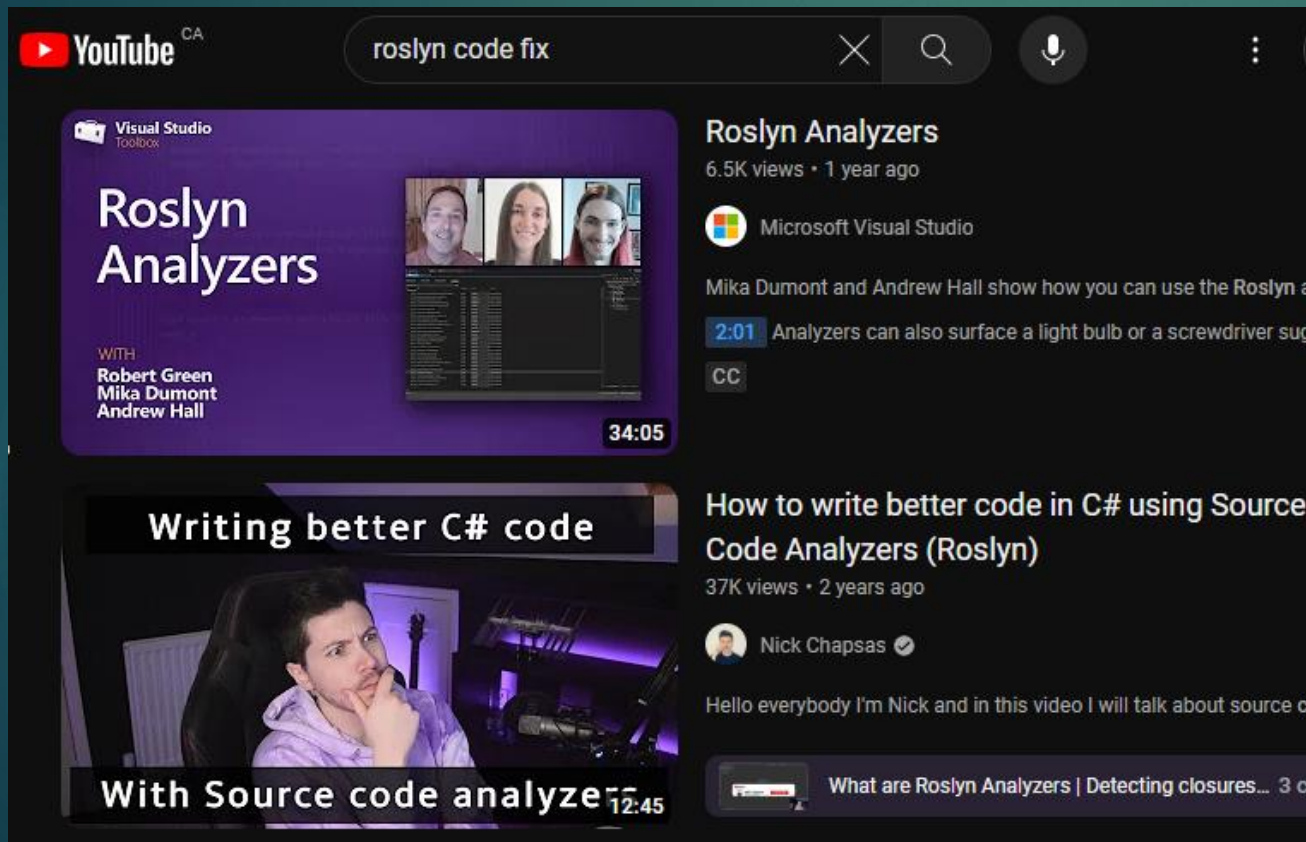
[View license](#)

[Security policy](#)

☆ 2.8k stars 🍴 230 forks 👁 38 watching 📄 Activity

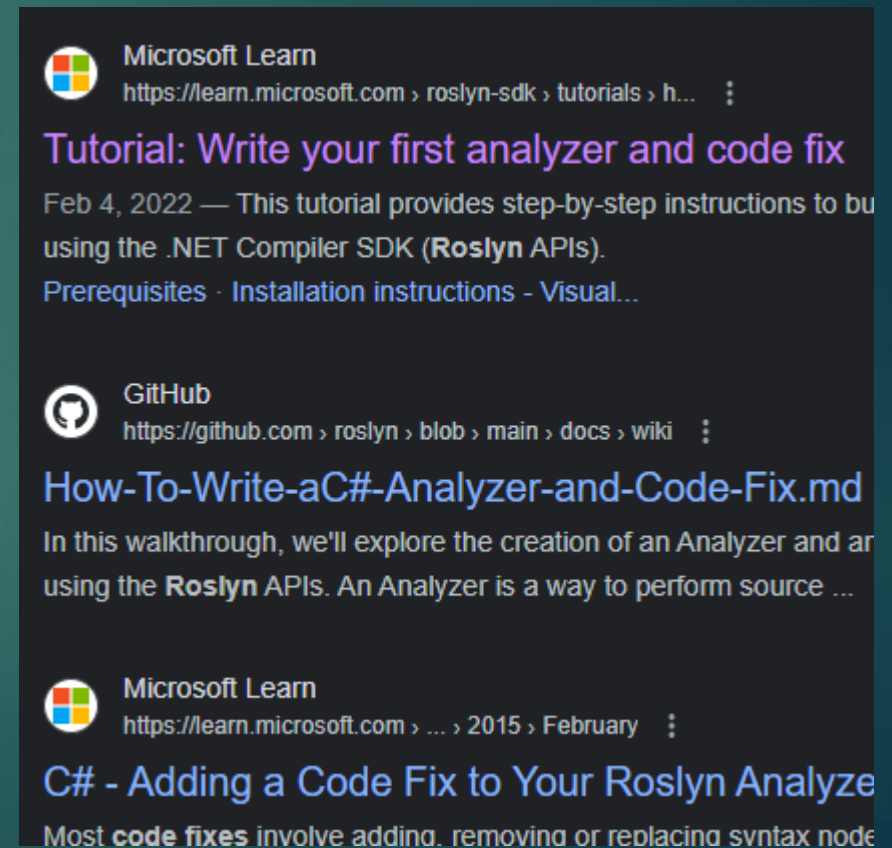
Our Code Fixes

There are tutorials and walk throughs on how to create our own.
It is not trivial work, but also not too bad.



The screenshot shows a YouTube search interface with the query 'roslyn code fix'. Two video results are visible:

- Video 1:** 'Roslyn Analyzers' by Microsoft Visual Studio. It has 6.5K views and was posted 1 year ago. The thumbnail shows three people (Robert Green, Mika Dumont, Andrew Hall) and a code editor. The video title is 'Roslyn Analyzers' and it is 34:05 long.
- Video 2:** 'How to write better code in C# using Source Code Analyzers (Roslyn)' by Nick Chapsas. It has 37K views and was posted 2 years ago. The thumbnail shows a man thinking. The video title is 'How to write better code in C# using Source Code Analyzers (Roslyn)' and it is 12:45 long.



The screenshot shows two web pages related to Roslyn code fixes:

- Microsoft Learn:** The page is titled 'Tutorial: Write your first analyzer and code fix'. It was published on Feb 4, 2022. The description states: 'This tutorial provides step-by-step instructions to build an analyzer and code fix using the .NET Compiler SDK (Roslyn APIs)'. Prerequisites include 'Installation instructions - Visual...'. The URL is <https://learn.microsoft.com/roslyn-sdk/tutorials/h...>
- GitHub:** The page is titled 'How-To-Write-aC#-Analyzer-and-Code-Fix.md'. The description states: 'In this walkthrough, we'll explore the creation of an Analyzer and a Code Fix using the Roslyn APIs. An Analyzer is a way to perform source code analysis...'. The URL is <https://github.com/roslyn/blob/main/docs/wiki/How-To-Write-aC#-Analyzer-and-Code-Fix.md>

Fin Costs...

Aren't that bad actually

- ▶ Workflow is a bit more complicated although not much compared to dual target testing with C++
- ▶ A bit slower when testing and iterating on actual hardware
 - ▶ Compile with fin
 - ▶ Compile with C compiler
 - ▶ Flash and test
- ▶ The project might fizzle out
 - ▶ You'll be left with a bunch of high quality C99
 - ▶ Easy to transpile all fin code to C++ as a migration strategy
 - ▶ C# Test code would need manual porting though
- ▶ Requires using C#
 - ▶ Not that bad - it is cross platform, free & open source
 - ▶ vscode is also free and has good C# support even on Linux

Rough Plan



Phase 0 – test the water

- ▶ Gather feedback

Phase 1 – bare minimum

- ▶ Simple classes
- ▶ Integer conversions
- ▶ Pointers, references
- ▶ C style arrays
- ▶ C style strings

Phase 2

- ▶ Function pointers
- ▶ Safe arrays
- ▶ Safe strings
- ▶ Const objects
- ▶ Volatile
- ▶ Interfaces
- ▶ Generics (good for `Queue<T>`)
- ▶ Inheritance
- ▶ Virtual methods
- ▶ Light weight exceptions

Phase 3

- ▶ C++ compatibility?
- ▶ `smartptr`?
- ▶ `constexpr`?

Interop

If fin code doesn't yet support a feature you need, you can always implement that part in C/C++.

Can I Really Do This?

Yes.

The uncertainties are really about how ergonomic the language will end up being in the tricky areas. Most areas are an ergonomic improvement (especially test).

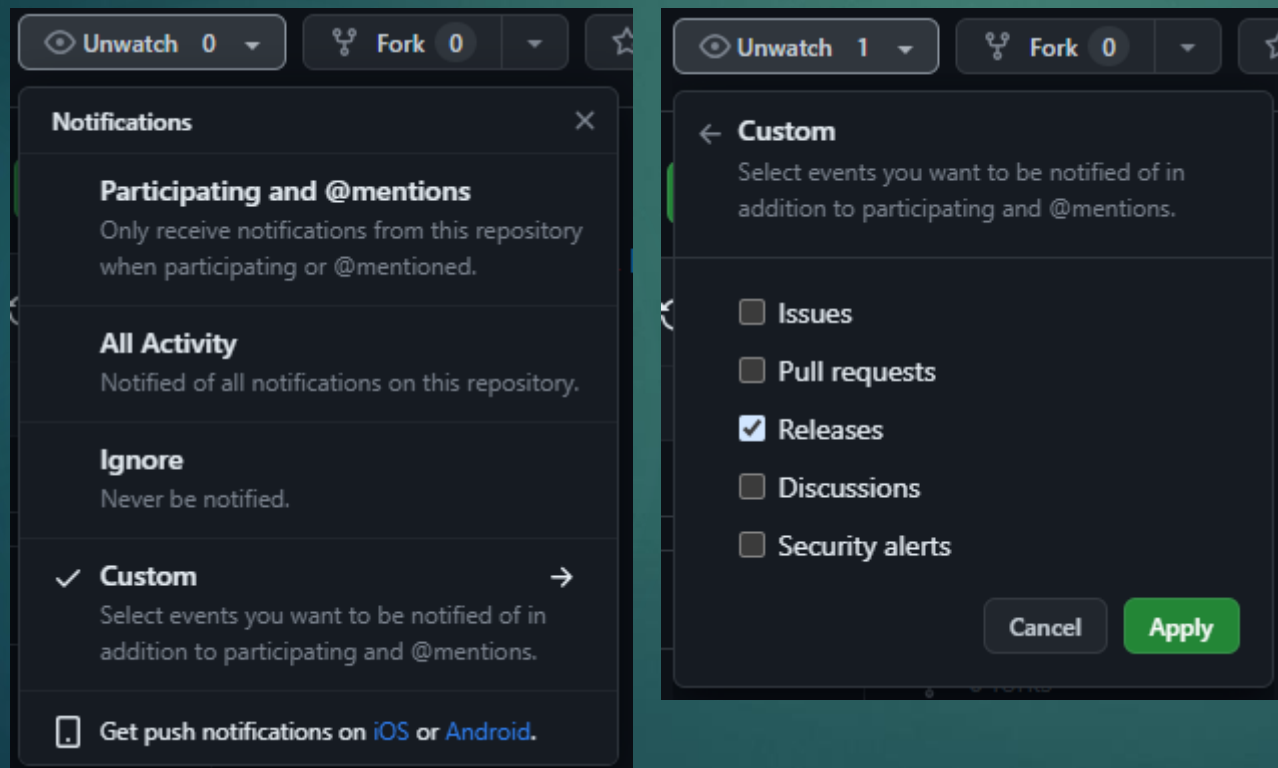
I have explored lots of the challenges and I finally think it's GO time!

StateSmith already translates a small part of C# to C.

How You Can Help

Provide (gentle) feedback. <https://github.com/fin-language/fin/>

Maybe watch project on github for releases...



Still Interested?

Thanks for your interest! Hope to see you on github :)

Write Embedded C
Faster, Better, Safer
With Fin

AN INTRODUCTION



Fin Language
Features

SOME OF THE GOOD STUFF



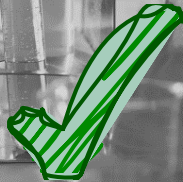
Multiple Controller
Simulations FTW

JAVA TO C99 (10 YEARS AGO)



Embedded C
Software Testing
& Design Choices

WITH ASSOCIATED CHALLENGES



Language
Comparison

NO LANGUAGES WERE HURT...



Fin Implementation
And Challenges

WE CAN DO IT!

