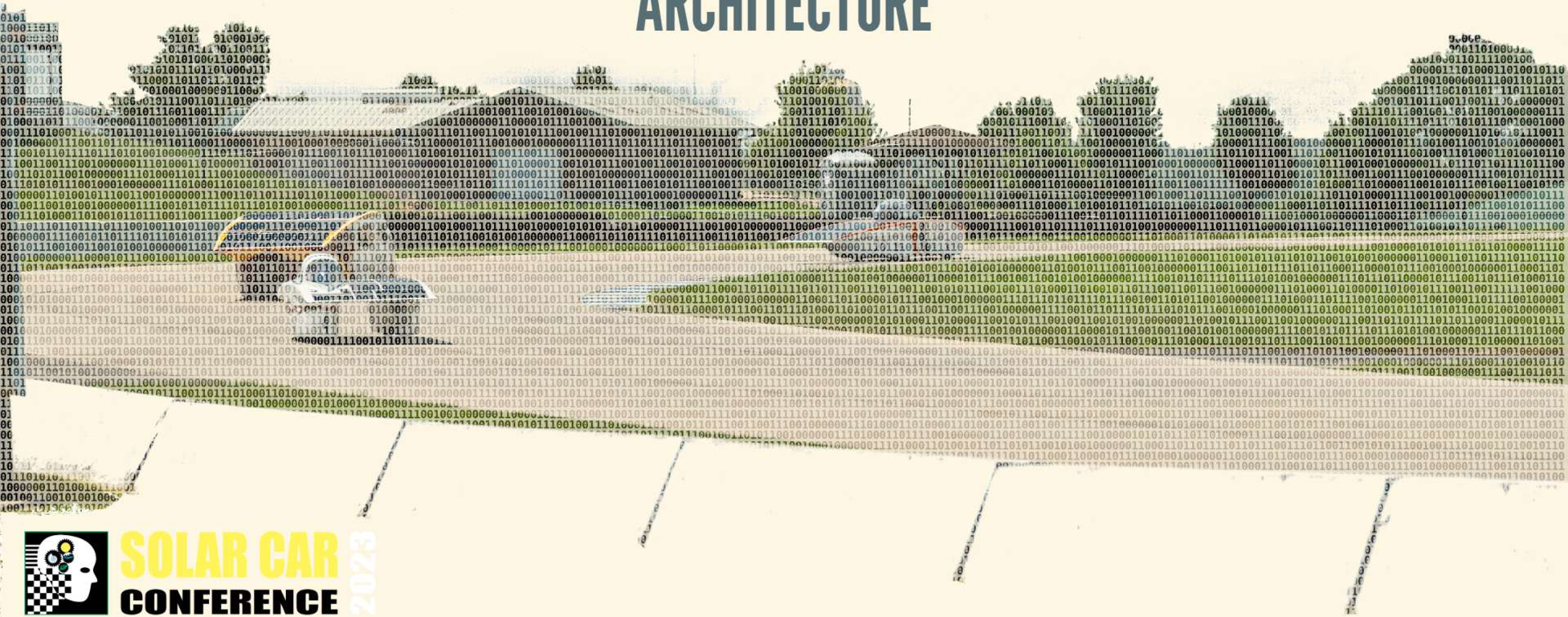
EMBEDDED SOFTWARE AND EMBEDDED SOFTWARE ARCHITECTURE



SOLAR CAR CONFERENCE 2023

JONATHAN MULLEN

ABOUT ME: JONATHAN MULLEN

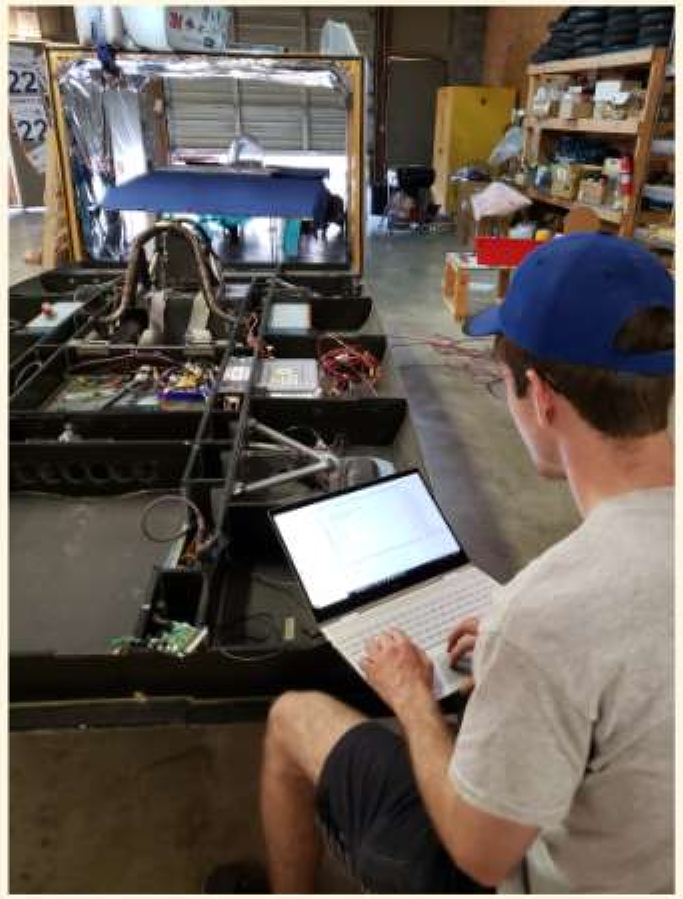
- B.S. Computer Engineering 2019 - University of Illinois
- Currently: Embedded Software Engineer @ Optivolt
- Previously: Embedded Software Engineer & Controls Engineer @ John Deere
- ASC 2022 Race Staff

SOLAR CAR CONFERENCE 2023

ME & SOLAR CAR

- Illini Solar Car Team
  - President, Business Lead, Electrical Co-Lead
  - Team Captain ASC 2018 & FSGP 2019
- Left: Argo (2017)
  - Wrote Code for Driver UI, BMS, & Lights
  - Driver in 2018 & 2019
- Right: Brizo (2021)
  - Desiged Steering Wheel Electrical Hardware
  - Wrote Code for Steering Wheel, BMS, Motor Control, Datalogger

SOLAR CAR CONFERENCE 2023



SOLAR CAR CONFERENCE 2023

TODAY'S OVERVIEW

- Embedded Firmware For Solar Cars
  - BMS Fault Prevention
  - Testing
- Embedded Firmware Architecture
  - Coupling (and modularity)
- Hardware Choices (briefly)
- Electrical System Architecture (briefly)

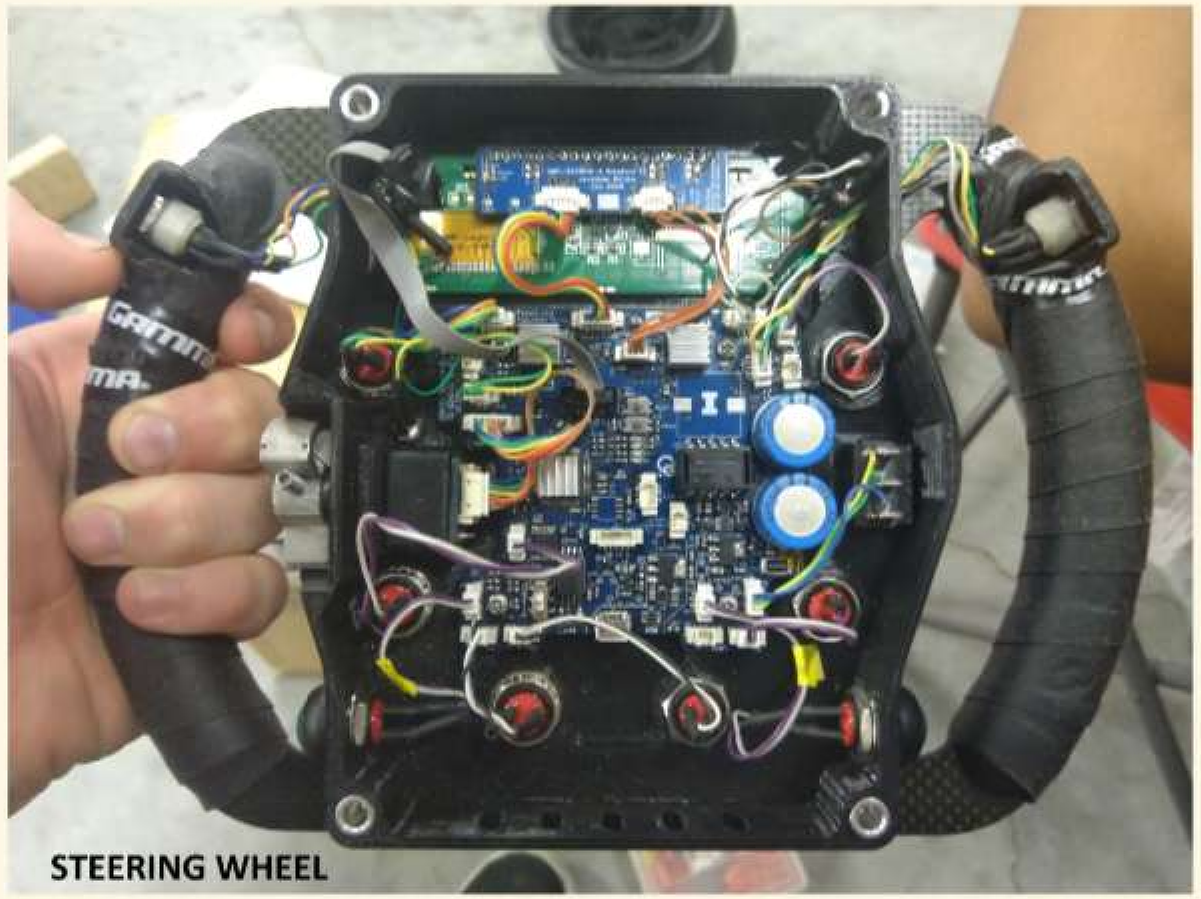
After: Question / Discussion Time

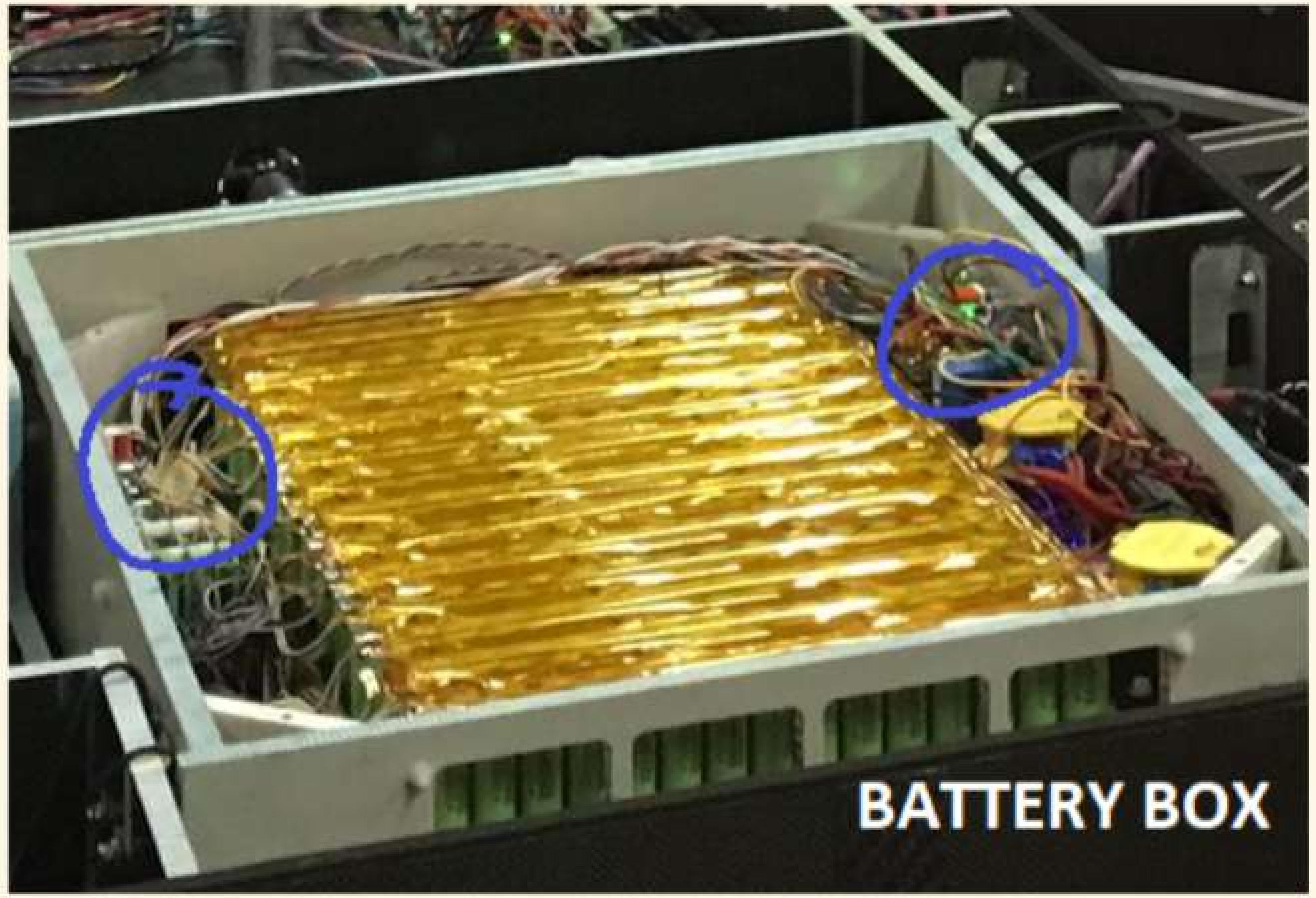
WHAT IS EMBEDDED SOFTWARE

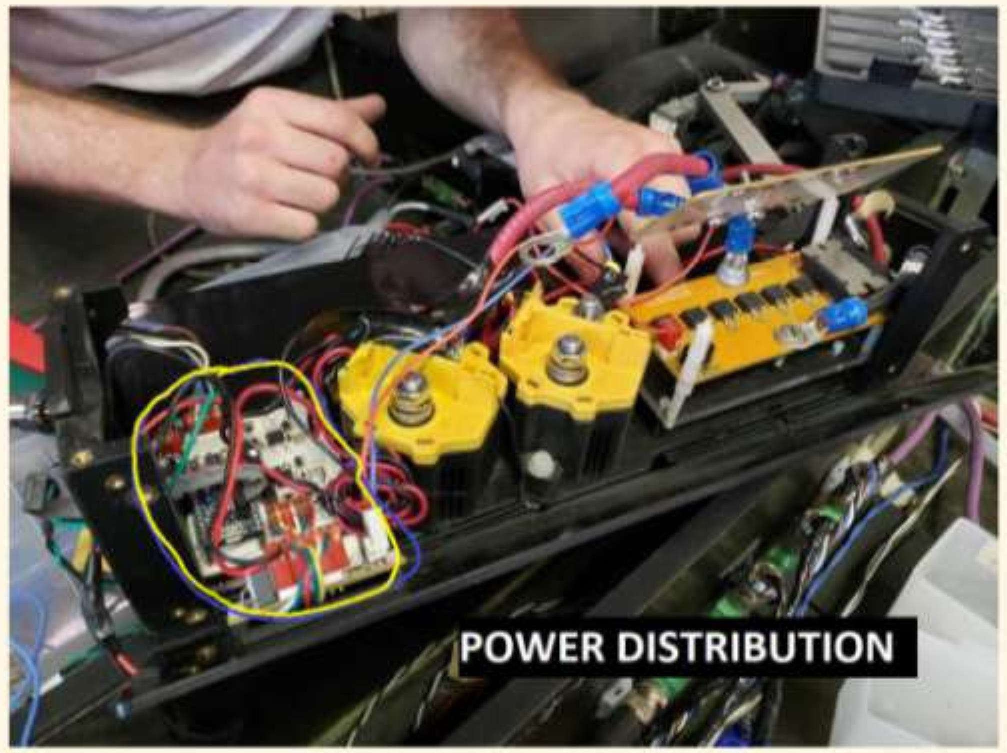
1. Software that interacts with specific hardware
2. On a microcontroller in a device not considered a computer

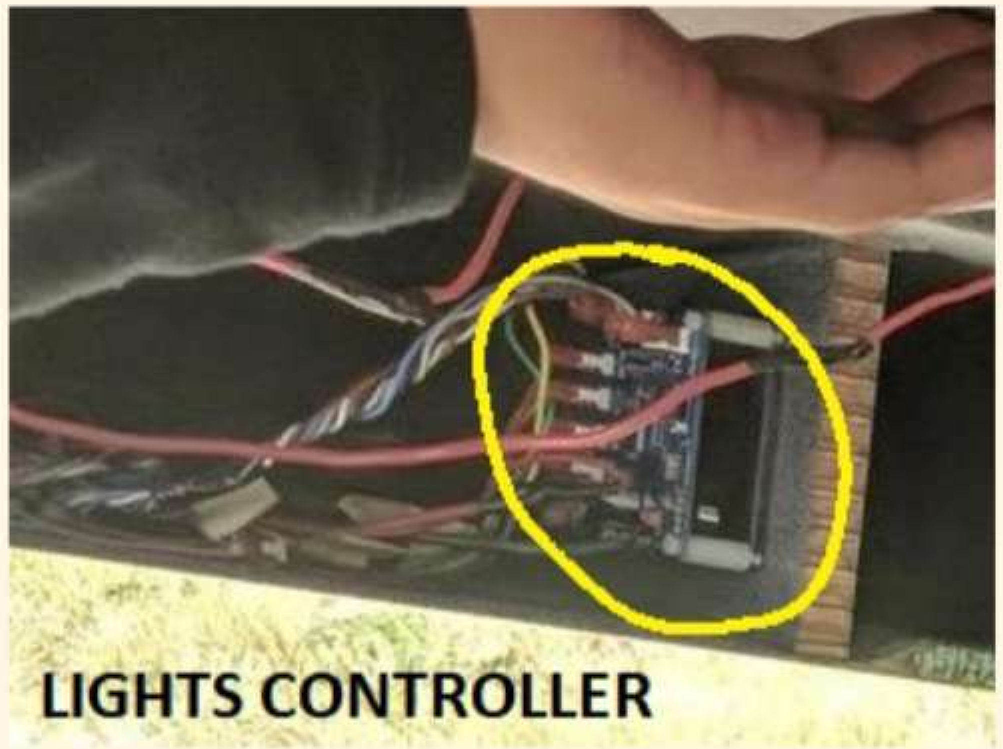
- Line between embedded software and firmware is *blurry*
- Almost always written in a compiled languages
- Software typically all stored in on-chip memory

EMBEDDED SYSTEMS IN SOLAR CARS


STEERING WHEEL


BATTERY BOX


POWER DISTRIBUTION


LIGHTS CONTROLLER

WHAT IS SOFTWARE ARCHITECTURE

AND WHY IS IT IMPORTANT

- Software Architecture is the design of your software
  - Designs the structure, interactions, and overall behavior of the system
- Good Software Architecture makes your life easier
  - Reduces bugs
  - Easier to Debug & Test
- Good Architecture helps your team perform better
  - Help you stay on the road / get back on the road faster if something goes wrong
  - Your codebase will be more maintainable & extendable

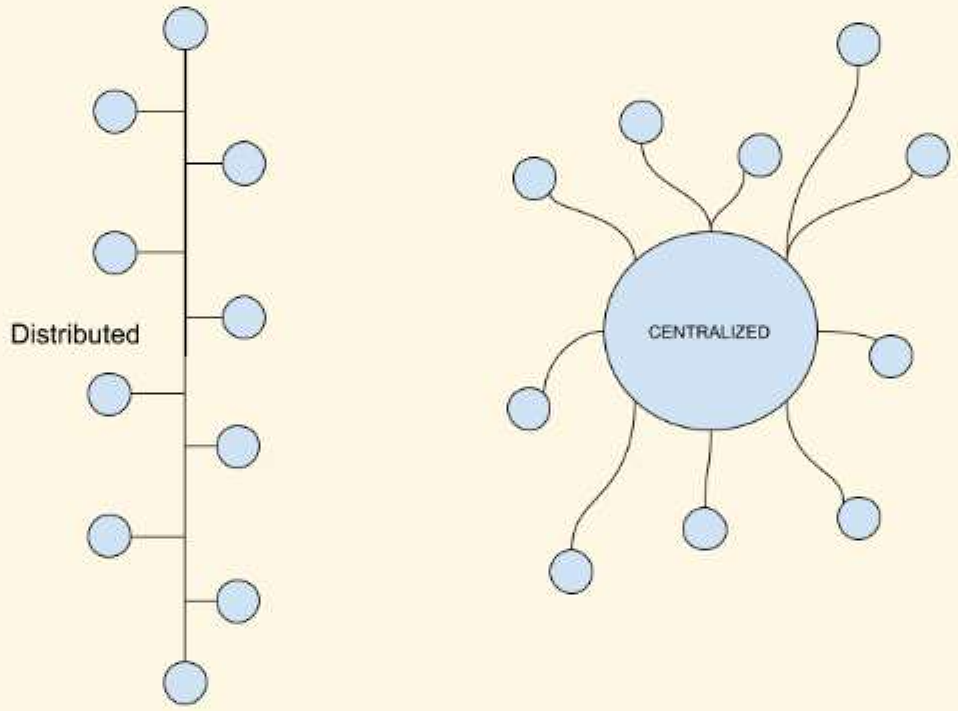
HARDWARE

- Think about software when making hardware choices!
- Overspeccing MCUs will save you lots of time & complexity
- Use the same (family of) MCU on all your boards!
- Specs / Features to Consider:
  - Memory & Speed
  - Built in Watchdog Timer, Brownout Detection, etc.
  - Floating Point Hardware
  - Reassignable Peripheral Pins
- At least look over the programming portion of datasheets before hardware design
  - Sometimes are gotchas that can lead to a hardware revision...

SYSTEM ARCHITECTURE

Distributed	Centralized
More HW & SW	More Complex HW & SW
Simpler & Less Coupled Sofware	Avoid Networking Complexities
Communications add delays	May require more software overhead
Failures Are Independent	Fewer Things to Fail
Often Easier to Add Functionality	Individual Parts have more complete Info

- Every solar car is somewhat distributed, balance how much



NETWORKING

- On (solar) cars CAN is most common
- Some Network Types (Especially CAN) are very difficult to analyze
  - For this reason, >30% utilization is often considered a full CAN bus
  - Beyond this it is hard to be sure low priority messages are on time (or close to on time)
- Generally I suggest using periodic messages, not request based, on solar cars
  - More Predictable
  - Don't stop getting data if requester has issues

FIRMWARE ARCHITECTURE

Hardware | HW Interface | Drivers | RTOS/Scheduler | Solar Car Logic

FIRMWARE ARCHITECTURE

Hardware | HW Interface | Drivers | RTOS/Scheduler | Solar Car Logic

- Microcontroller (MCU)
  - GPIO, ADC, Timers, Communications, etc.
- Peripheral Hardware
  - IO Expander, Encoders, Screens, etc.

FIRMWARE ARCHITECTURE

Hardware | **HW Interface** | Drivers | RTOS/Scheduler | Solar Car Logic

- Handles MCU Bring Up, configuration
  - If used, an OS will handle much of this
- Includes any code that writes MCU registers
- Exposes an API for hardware interaction

FIRMWARE ARCHITECTURE

Hardware | HW Interface | **Drivers** | RTOS/Scheduler | Solar Car Logic

- Interfaces with peripheral hardware
  - Sometimes provided by manufacturers
  - Likely uses the HW Interface
- Or, creates a virtual device such as file system

FIRMWARE ARCHITECTURE

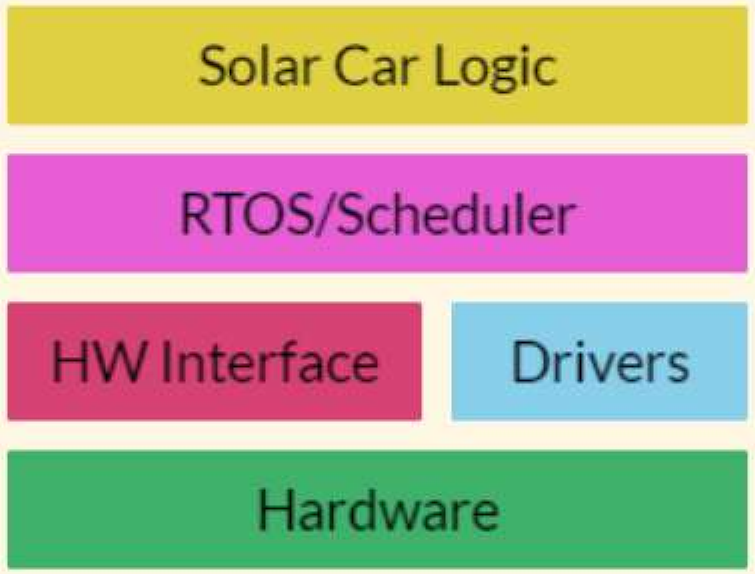
Hardware | HW Interface | Drivers | **RTOS/Scheduler** | Solar Car Logic

- Determines what happens and when
  - Must service hardware and peripherals as required
  - All "solar car logic" called from here
- RTOS not required for a solar car

FIRMWARE ARCHITECTURE

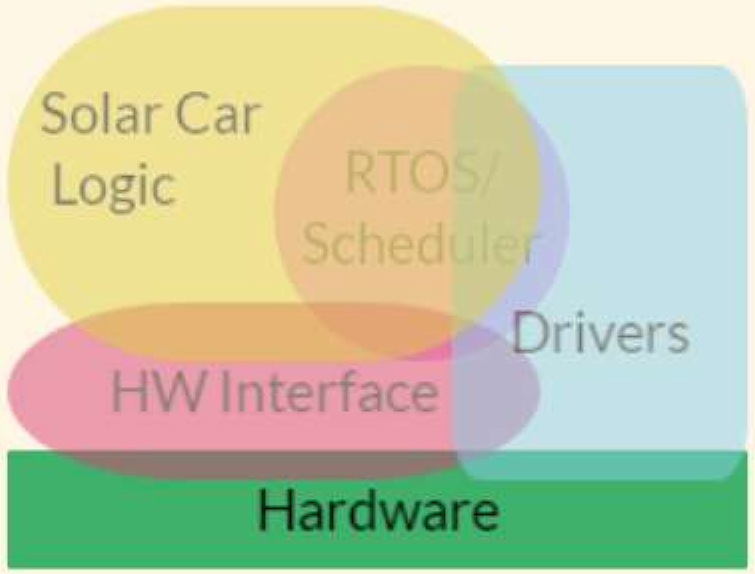
Hardware | HW Interface | Drivers | RTOS/Scheduler | **Solar Car Logic**

- The "business logic"
- Makes it work like a solar car
  - Interactive: Acceleration, Regen Braking, Lights, Displays
  - Background: BMS, MPPT, Datalogging, Telemetry



FIRMWARE ARCHITECTURE

THE GOAL



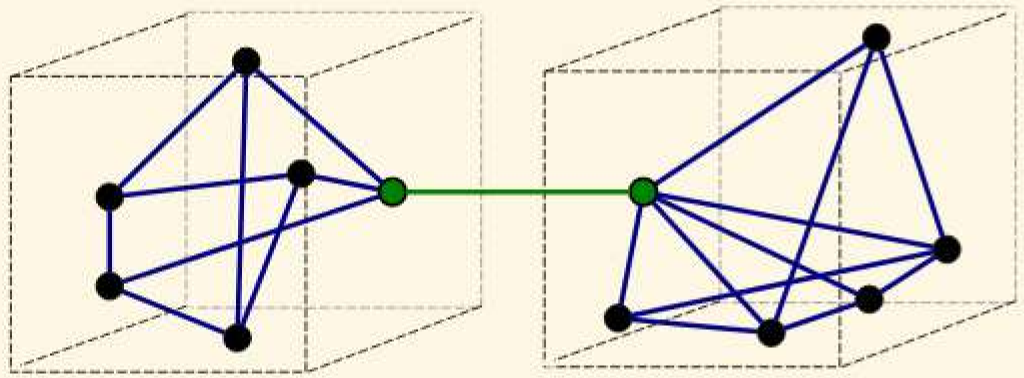
FIRMWARE ARCHITECTURE

WHAT HAPPENS

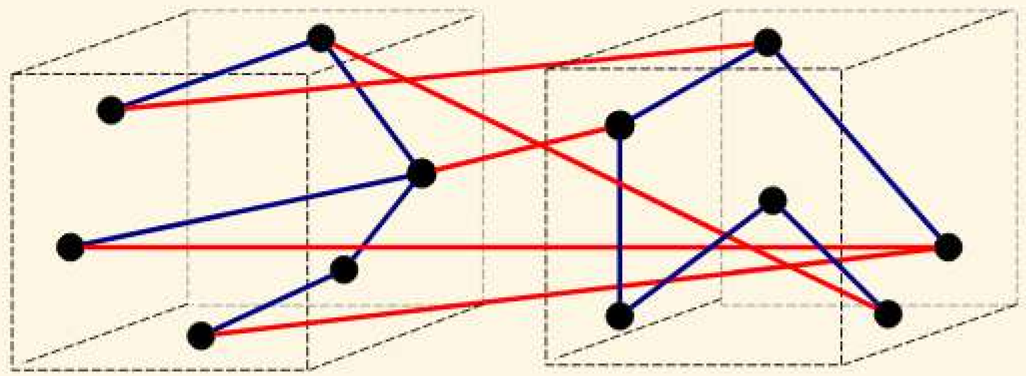
COUPLING & COHESION

GENERALLY INVERSELY RELATED

- Coupling is how two systems interact with each other
- Cohesion is how the parts within a system are related
- Goal is High Cohesion & Low Coupling
- One does not always follow the other, but usually does
- For our purposes, we will assume an inverse relationship



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

HIGH COUPLING

THE CONSEQUENCES

- Correlated With More Bugs[Shown in a real study!]
- Harder to make changes
- Slower to make changes
  - Need to understand more to make changes
  - Easier to break unrelated things!

LOW COUPLING

THE BENEFITS

- Features & Systems can be tested independently
- If one part is broken, it can be easily removed/disabled
- When Hardware Changes easy to replace individual pieces as needed
- When Requirement Change easy to replace individual pieces as needed

GOOD ARCHITECTURE: AVOIDING HIGH COUPLING

1. GOOD REQUIREMENTS

- Start High Level With the System and Work Down to Specifics
    - This can help with system architecture too
- Requirements come from many places
    - Competition (or government) Regulations
    - Purchased Components
    - Competitive Uses
    - Basic Vehicle Functionality
    - Your Team's Goals / Desires

GOOD ARCHITECTURE: AVOIDING HIGH COUPLING

2. GOOD SPECIFICATIONS

- Specifications are Specific and Technical
  - Whereas Requirements are Broad and Descriptive
  - Each Requirement likely turns into multiple specifications
- Describes the solution to meet the requirement
- Should be explicit - specifications should imply needs

GOOD ARCHITECTURE: AVOIDING HIGH COUPLING

3A. BLOCK DIAGRAMS

- Turn your requirements & corresponding specifications into blocks
- Each block should fit into only one of the categories
  - Hardware
  - Hardware Interface
  - Peripheral Drivers
  - RTOS / Scheduled
  - Solar Car Logic

GOOD ARCHITECTURE: AVOIDING HIGH COUPLING

3B. GOOD INTERFACE DESIGN

- Yes, interfaces internal to the firmware should be designed
- Map out data that needs to be shared from component to another
- Decide which component owns the data - minimize global data
- Use getter/setter functions with minimal side effects
- Avoid designs that result in neccessary sequences of calls

GOOD ARCHITECTURE: AVOIDING HIGH COUPLING

4. SEPARATING CODE REVIEW FROM DESIGN REVIEW

- Have your architecture sorted out before you start writing code
- Doing and reviewing design and implementation at the same time is messy
- You will take shortcuts for short-term easy implementation that can cause you headaches when something changes
- Have Software Design & Code Reviews

ARCHITECTURE WALKTHROUGH

OVERVIEW

- Goal is to replace the analog controls on a Mitsuba 1kW Motor Controller
- System needs to receive info via CAN
- Output to Motor Contoller via Peripherals

ARCHITECTURE WALKTHROUGH

1. GOOD REQUIREMENTS

- *Driver Inputs Transmitted via CAN*
- *Torque Output of Motor Controller Scales with Accelerator Pedal Press*
- *Motor Cannot Output Positive Torque when brake pedal pressed*
- *Regen Braking Enabled By Buttons Regen pedal*
- *Regenerative Braking Limited to Difference Between Max Charge Current and Solar Array Output to Avoid Overrcurrent Faults (with a margin)*
- *Regenerative Braking Power Reduced at High Voltages to avoid over voltage faults*
- ...

ARCHITECTURE WALKTHROUGH

2. GOOD SPECIFICATIONS

- *Regenerative Braking Limited to Difference Between Max Charge Current and Solar Array Output to Avoid Overrcurrent Faults (with a margin)*
  - ***Total Current to be limited to margin of 0.5A below cutoff of 25A***
  - ***Receive Solar Current from MPPTs via CAN Message***
  - ***Must Limit within 0.25s to avoid shutdown fault***
- *Regenerative Braking Power Reduced at High Voltages to avoid over voltage faults*
  - ***Regen Current Limited to current that results in high cell voltage of 4.15V***
  - ***Voltage Rise Calculated based on equal rise across all cells with pack ESR of 125mΩ***
  - ***Must Limit within 0.25s to avoid shutdown fault***

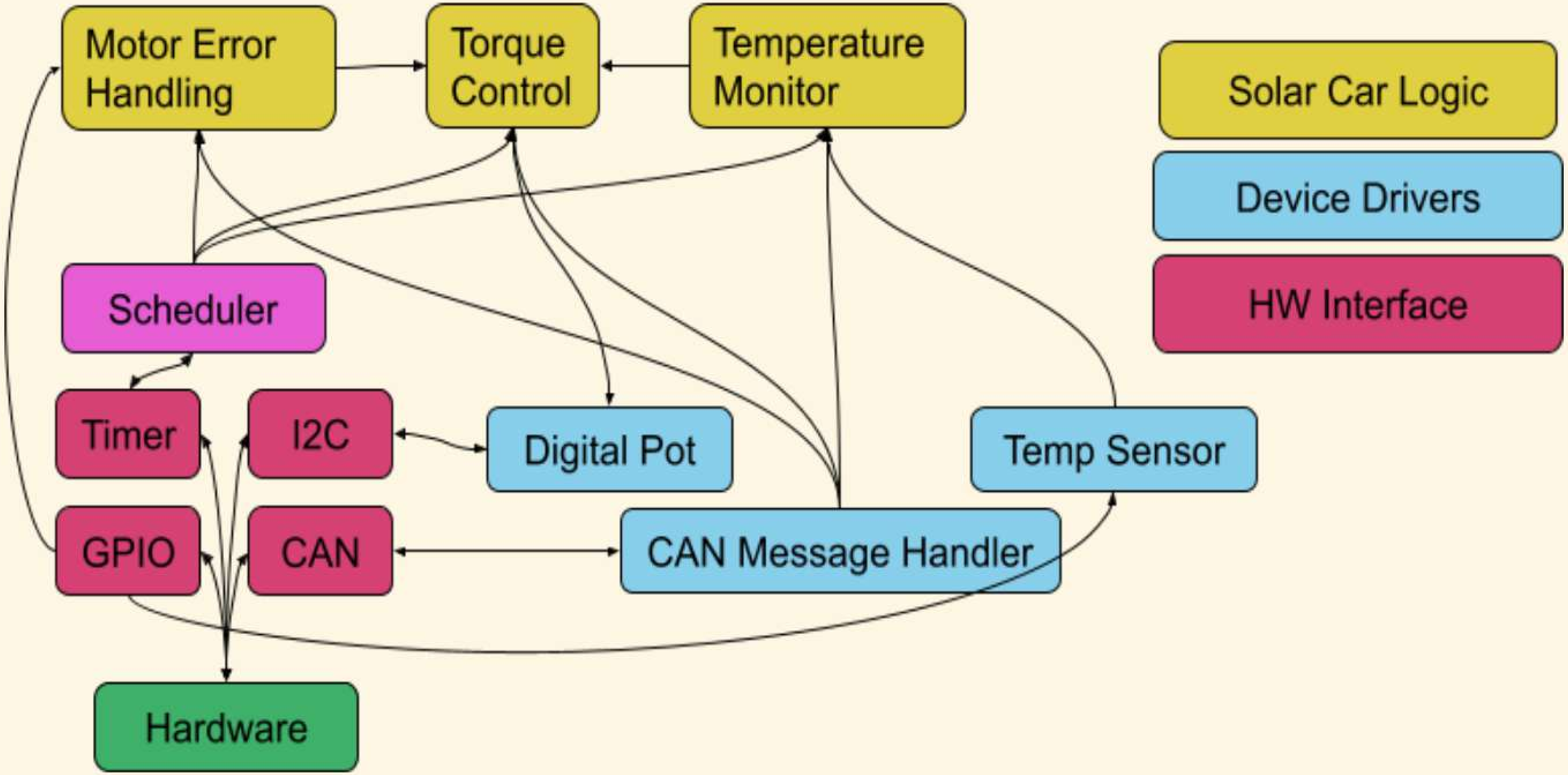
ARCHITECTURE WALKTHROUGH

2. GOOD SPECIFICATIONS

- *Regenerative Braking Limited to Difference Between Max Charge Current and Solar Array Output to Avoid Overrcurrent Faults (with a margin)*
- *Regenerative Braking Power Reduced at High Voltages to avoid over voltage faults*
  - ***Receive Solar Current from MPPTs via CAN Message***
  - ***Receive Max Charging Current (in 1/10A) Allowed from BMS via CAN***
  - ***Must Limit Current within 0.25s to avoid shutdown fault***
  - *~~Total Current to be limited to margin of 0.5A below cutoff of 25A~~*
  - *~~Regen Current Limited to current that results in high cell voltage of 4.15V~~*
  - *~~Voltage Rise Calculated based on equal rise across all cells with pack ESR of 125mΩ~~*

ARCHITECTURE WALKTHROUGH

3A. BLOCK DIAGRAMS



ARCHITECTURE WALKTHROUGH

3B. GOOD INTERFACE DESIGN

Torque Control Class

- `InitializeTorque(Digital Potentiometer)` : Initialize Output to 0, inputs to worst case
- `PedalPositions(CAN Message)` : Sets the Pedal Positions and Updates Torque Output
- `BMSCurrentLimit(CAN Message)` : Sets BMS Current Limit and Updates Torque Output
- `SolarArrayCurrent(CAN Message)` : Sets Solar Array Current and Updates Torque Output
- `MotorErrorState(MotorError)` : Sets motor error state and and Updates Torque Output
- `MotorTemperatureDerate(Derating)` : Sets torque derating and and Updates Torque Output

ARCHITECTURE WALKTHROUGH

3B. GOOD INTERFACE DESIGN

Torque Control Class (revised)

- `InitializeTorque(Digital Potentiometer)` : Initialize Output to 0, inputs to worst case
- `PedalPositions(CAN Message)` : Sets the Pedal Positions ~~and Updates Torque Output~~
- `BMSCurrentLimit(CAN Message)` : Sets BMS Current Limit ~~and Updates Torque Output~~
- `SolarArrayCurrent(CAN Message)` : Sets Solar Array Current ~~and Updates Torque Output~~
- `TorqueControlTask()` : Calculate New Torque and Write to Potentiometer
- ~~`MotorErrorState(MotorError)` : Sets motor error state and and Updates Torque Output~~
- ~~`MotorTemperatureDerate(Derating)` : Sets torque derating and and Updates Torque Output~~

ARCHITECTURE WALKTHROUGH

4. CODE & DESIGN REVIEWS

- Design Reviews Along the Way At Specified Checkpoints
  - As We've been doing!
- Use Review Checklists
- Code Review Checklist Might Include:
  - Obvious Bugs
  - Does it Meet Requirements & Specifications
  - Does it implement the Interface As Described
  - Well Commented & Readable
  - Internal Functions are well designed, no duplication
  - All Possible Return/Error Values Handled
  - Thread Safety, Scopes Minimized, Memory Allocation Safety, etc

WRITING THE CODE

TORQUE CONTROL

```c
/*
 * @file torque_control.c
 * @brief Controls Torque Signal to Motor Controller based on
 *            CAN Data & Other Inputs
 */

//Internal Variables
DigitalPot* pot; // Digital Potentiometer object
unsigned BMSLimit_mA = 0; // Current Limit from BMS
unsigned MPPTCurrent_mA = -1; // Current Provided by MPPTs
bool brakePressed = true; // Track if brake is pressed
float accelPedal = 0; // Position of the Accelerator Pedal
float regenPedal = 0; // Position of the Regen Pedal

// Update the Torque Values and Write to Potentiometer
```

WRITING THE CODE

TORQUE CONTROL

```
// Update the Torque Values and Write to Potentiometer
void torqueControlTask(){
  float accelTorque;
  float regenTorque;

  // If there is a motor error output 0 Torque
  if(getMotorError()){
    accelTorque = regenTorque = 0;
  }else{
    // Determine Acceleration Torque from Pedal
    // If brake pedal accel must be 0
    if(brakePressed){
      accelTorque = 0;
    }else{
      accelTorque = accelPedal / 1.0 * MAX_TORQUE;
```

WRITING THE CODE

TORQUE CONTROL (REVISED)

```
// Update the Torque Values and Write to Potentiometer
void torqueControlTask(){
  float accelTorque;
  float regenTorque;

  // If there is a motor error output 0 Torque
  if(getMotorError()){
    accelTorque = regenTorque = 0;
  }else{
    // Determine Acceleration Torque from Pedal
    // If brake pedal accel must be 0
    if(brakePressed){
      accelTorque = 0;
    }else{
      accelTorque = accelPedal / 1.0 * MAX_TORQUE;
```

WRITING THE CODE

SCHEDULER (SIMPLEST)

```
int main(){
  // Setup Code Here

  while(1){
    // Get the Current Time
    currentTime = timer.now();

    // CAN Task
    if(CAN.bufferNotEmpty()){
      processCANMessages();
    }

    // Torque Task
    if(now - lastTorqueTime > TORQUE_TASK_RATE){
      torqueControlTask();
      lastTorqueTime = now;
    }

    // Motor Error Monitor Task
    if(now - lastErrorTime > ERROR_TASK_RATE){
      motorErrorTask();
      lastErrorTime = now;
    }
  }
}
```

This requires discrete, small, & fast tasks with few variations in behavior

WRITING THE CODE

SCHEDULER (STILL SIMPLE)

```
int main(){
  // Setup Code Here

  while(1){
    // Get the Current Time
    currentTime = timer.now();

    // CAN Task
    if(CAN.bufferNotEmpty()){
      processUpTo10CANMessages();
    }

    // Torque Task
    if(now - lastTorqueTime > TORQUE_TASK_RATE){
      torqueControlTask();
      lastTorqueTime = now;
      continue;
    }

    // Motor Error Monitor Task
    if(now - lastErrorTime > ERROR_TASK_RATE){
      motorErrorTask();
      lastErrorTime = now;
      continue;
    }
```

This can help if you have some slow tasks to keep more important tasks running closer to on time

WRITING THE CODE

SCHEDULER (RTOS)

```
int main(){
  // Setup Code Here

  // Launch Tasks
  startTask(torqueControlTask, TORQUE_TASK_RATE, TORQUE_TASK_PRIORITY);
  startTask(motorErrorTask, ERROR_TASK_RATE, ERROR_TASK_PRIORITY);


  while(1){

    // CAN Task
    if(CAN.bufferNotEmpty()){
      processUpTo10CANMessages();
    }
  }
}
```

More complex to set-up, need to worry about everything getting interrupted, but will meet your time requirements more precisely and potentially get more out of your hardware

DESIGN FOR RELIABILITY

- Embedded Code on a Solar Car is about reliability and maximizing your car - not the code
- Minimum Goal is Keep the Car on the Road:
  - Consider how things will go wrong, handle recoverable failures
    - Bounds Checking - ignore obviously bad data
  - Make it easy to "turn off" non-bare neccessities
  - Minimize assumptions - confirm as much as possible
    - If you must, document it
  - Handle Rolling Resets Where allowable
    - Use Watchdog Timers
  - Be prepared to lose power at any time

TESTING YOUR CODE

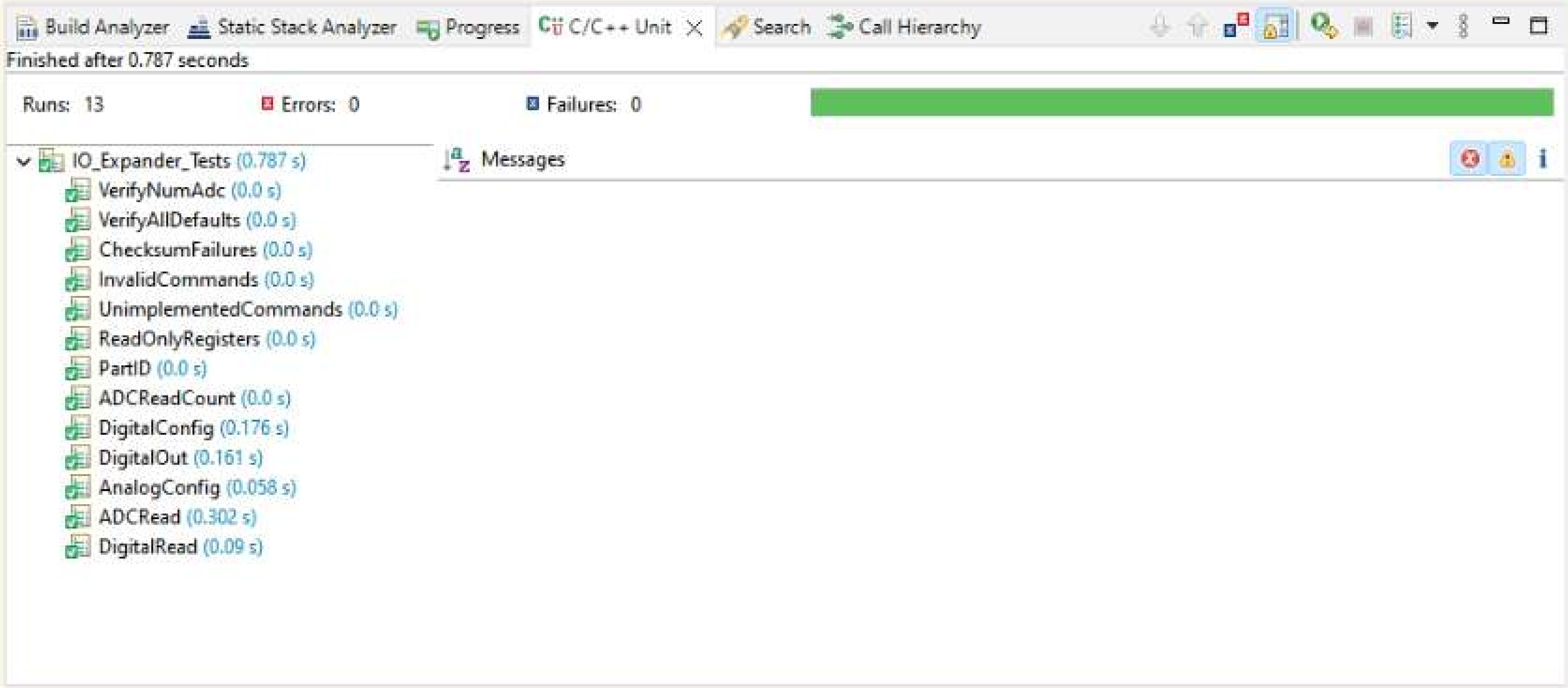
UNIT TESTING

- Tests your code on your computer!
- Very Useful to catch logic bugs and to keep working things working
- Can help make sure there are no cases in which it fails, instead of checking that there is one in which the code succeeds
- Recommend doing at the very least for any complex state machines or data manipulation
- Will require mocks to fake any external/hardware calls
- Libraries such as Google Test (with Google Mock), Unity Test (with C Mock) , CppUTest (with CppUMock) can make this easy
    - Many Have Integrations with your favorite IDE

TESTING YOUR CODE

DESIGN FOR UNIT TEST

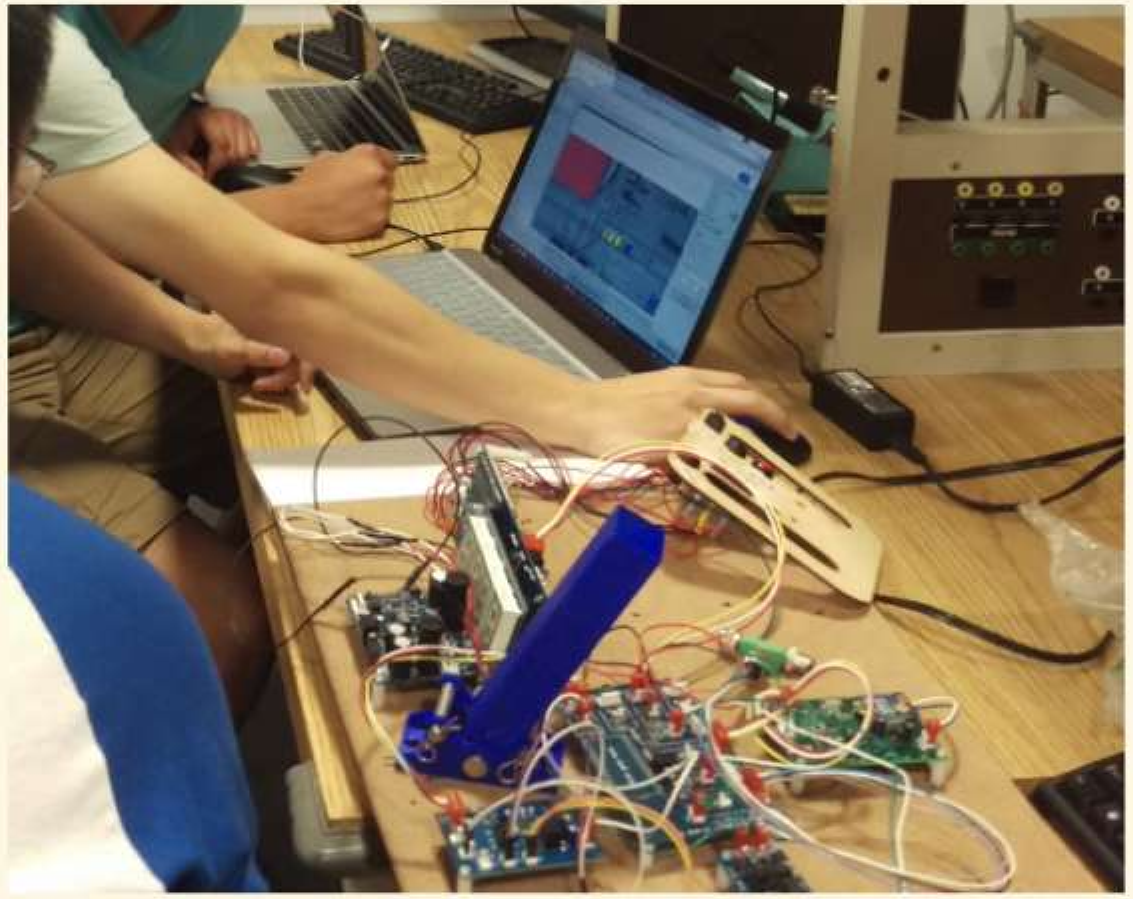
- Testable Code is Small Pieces with High Cohesion
- Following the architecture step we discussed will make your code more testable
- Good Specifications will inform your tests
  - They specify what behavior is important to test
  - Provide a source of truth about desired behavior

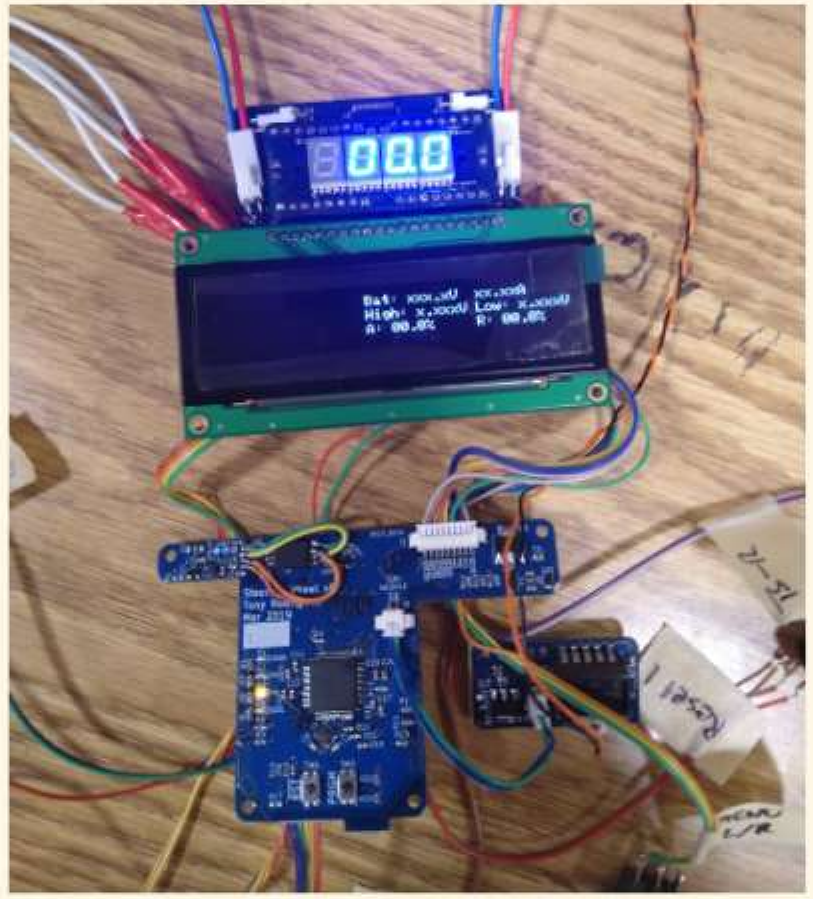


TESTING YOUR CODE

ON TARGET TESTS

- Automating this is possible, although value is debatable for a solar car team
  - Hardware in the loop testing (HIL Testing)
- Write Test Plans - make it repeatable for when you change things!
- Start by simulating inputs and validating outputs - send CAN messages, press buttons, etc.
- Try to break it! Have someone else try to break it!
- Debugging piece by piece will make it easier to isolate issues
- Invest in testing tools - CAN Bus analyzers, extra buttons, breakout boards, etc





TESTING YOUR CODE

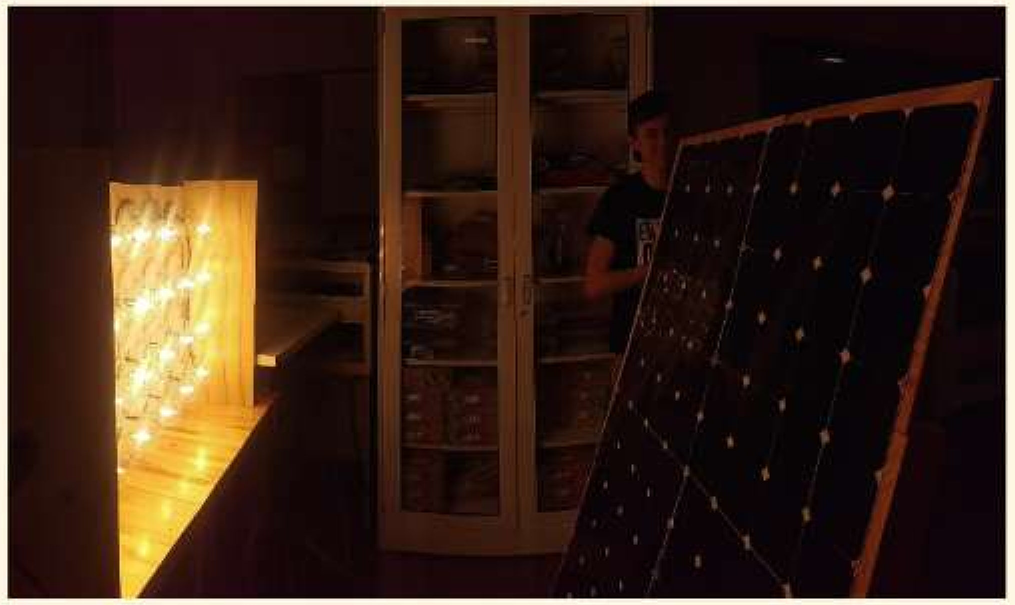
DESIGN FOR BENCH & CAR TEST (& RACE)

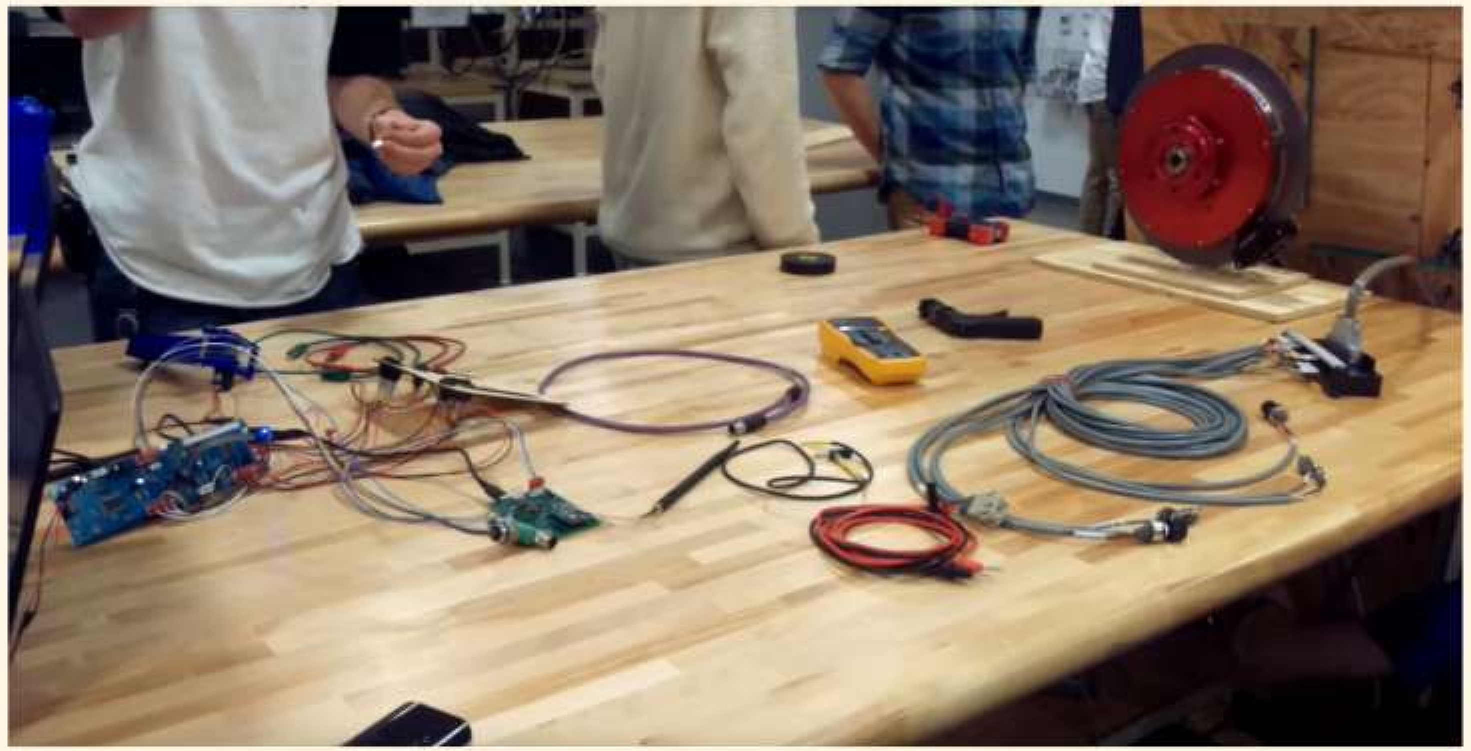
- Step 1. Get your telemtry, or at least logging, working
- Put Debug Data on CAN, more than you think you need
  - You can make it disableable or remove later
- Make Parameters Configurable without reprogramming (over CAN, better: Telemetry)
  - For example, when testing cruise control PID Controller, gains should be configurable without repogramming - significant time saver
  - Screen Brightness, maybe it will need to be brighter where the race is held
  - Feature Toggles for non-critical features - often better to stay on the road and just deal with it

TESTING YOUR CODE

SYSTEM BENCH TESTS

- Again, use test plans
- Piece by Piece assemble you system on the bench, testing some interactions and simulating others
- Eventually you want your whole system working on the bench, easier to fix than on the car
- Sometimes code bugs are time based, play with it for a while!





TESTING YOUR CODE

TEST ON THE CAR

- Repeat what you did on the bench!
- Do it ASAP, whole car doesn't need to be done!
- Test plan is important - safety risks anytime on the car
- When adding new features, run without first, then introduce it
- Sometimes code bugs are time based, play with it for a while!
    - Make sure the code can run a full day before you come to the race!
    - Or, for some systems make sure they can handle running restarts

THANK YOU!

QUESTIONS, OR RELATED DISCUSSION TOPICS WELCOME

Sources:
"Firmware architecture in 5 easy steps," Embedded.com, 28-Sep-2022. [Online]. Available: https://www.embedded.com/firmware-architecture-in-five-easy-steps/. [Accessed: 27-Jan-2023].

M. D'Ambros, M. Lanza, and R. Robbes, "On the relationship between change coupling and software defects." [Online]. Available: https://users.dcc.uchile.cl/~rrobbes/p/WCRE2009-changecoupling.pdf. [Accessed: 27-Jan-2023].

M. Dunn, "Toyota's killer firmware: Bad design and its consequences," EDN, 03-Apr-2020. [Online]. Available: https://www.edn.com/toyotas-killer-firmware-bad-design-and-its-consequences/. [Accessed: 27-Jan-2023].

By Евгений Мирошниченко - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=104043458

SOLAR CAR CONFERENCE 2023