Interfaces, Integration, and Incremental Development

Based on a lecture by Prof. André DeHon
(System-on-a-Chip Architecture, ESE 5320, Fall 2023)

Vishnu Venkatesh

Penn Engineering
University of Pennsylvania

Big question

- How do you think about and break down complex engineering projects?

Message

- Develop, test, and integrate in parallel
- Focus on interfaces early
- Start with something simple and then incrementally refine parts
  - May lack features or give you the wrong outputs initially
  - … but sets you up to get things done more efficiently

Common yet flawed approach

- Not thinking about the entire system during development
- Hoping that things will “figure themselves out” after you get individual components working
- Spend time (over)optimizing each component
  - My most common mistake
- Test each individually
- Integrate for the first time near the deadline

Fundamental Problem

- What happens to components – hardware or software – that you don't integrate till the very end?
  - Even if you get pieces working alone, what might happen when you try to put things together?
- Why is it natural to do this?
- Is there a better approach?

Recommended Approach

- Break up the system into interlocking components
- Focus on how components interact
- Simplify the functionality of each component to test out your interfaces
- Use placeholders for components that you haven't developed (software) or haven't arrived (hardware)
- Get as many components working together as you can
  - Cut corners, it's okay if it's the wrong functionality to begin with as long as the interfaces are correct
- Plan out how to refine each component while keeping the interfaces intact

Benefits

- You can be more confident that things will work when you put them together.
- Improves teamwork – development, testing, debugging can be done in parallel
- Debugging is easier, you have clear boundaries to localize errors
  - To narrow down on where exactly the error is, check your interfaces and see if data is being passed along as you expect
- Confidence boost of having something working, even if it needs refinement
- Supports work in small bursts.
- Scales effectively – these principles are applicable to projects of all sizes and complexity
- You can start working on firmware before you get your hardware
  - There will always be some part of your code that doesn't depend on the exact hardware or can be developed on a dev board.
  - By defining interfaces clearly, you can get started on that part and test it out – and then integrate it once your hardware arrives.
- Conclusion: Returns to efforts ratio checks out

Interfaces

- How different components of your system talk to each other
- Hardware
  - Communication protocols. Triggers, interrupt mechanism, response. Sensor outputs. How it's sent to the micro. Actuator control inputs.
- Software
  - Qualitative information to be shared between functions. What information do the numbers in the input/output data represent?
  - Data formats. Size? Form?

Firmware interfaces

- Much more flexibility than hardware interfaces
- Key principle: Information hiding
  - Wrap the details that are likely to change in a module
  - The interface, if thought through correctly, is less likely to change
- Information hiding: “To keep the interfaces “beautiful”, one needs to structure the system so that all arbitrary or changeable or very detailed information is hidden.”
  - - D. L. Parnas
  - Aim: to eliminate invisible dependencies between your modules

The secret history of information hiding | D. L. Parnas

Information Hiding - Embedded Artistry

Simplifications

- The bare minimum you need to test your interfaces
- Handle simple, specific subset of cases
  - Don't worry about performance
  - Don't worry about corner cases
- A placeholder for the actual functionality
  - Minimal support to begin with
- Test placeholder with sample inputs/tests for simple cases
  - It's okay if your outputs are wrong so long as they have the right form – improve on it iteratively.

Component simplification example

- Task:
  - You're making a battery management system.
  - Battery SoC and SoH algorithm needs one voltage and one current value every second.
  - You are getting voltage and current values from two ADCs simultaneously at 100 samples per second over SPI.
  - Assume that you already have a driver to read SPI data and write it into a circular buffer of 100 elements.
- Component?
  - Code that "condenses" the buffer contents and passes it to the algorithm.
- Interfaces?
  - Input: Circular buffer
  - Output: One floating point value

Component simplification example (contd.)

- What is the simplest possible placeholder for this component?
  - Function that returns the first value in the buffer
  - Is the form of the output correct? Does it match the expected interface?
  - Is the encapsulated part (how you calculate the output) something that is likely to change?
  - Is the interface likely to change?
- You might get poor results to start with
  - … but you can later replace the placeholder with another one that works better
- Iteratively make it more sophisticated
  - Simple average
  - Exponential filter
  - Complicated DSP stuff: Hamming/Blackman windows, Chebyshev/Butterworth filters
  - …anything else so long as it fits the interface
  - Maybe different filters work better for voltage and current – fixed interfaces make it easy to iterate on them separately

```
float filter(uint8_t* cbuf)
{
    return (float)cbuf[0];
}
```

Two functions like this, one for voltage and one for current

Challenges

- Requires a bit of upfront time investment
  - But saves time during development, so it's worth it
- Many ways of doing it
  - Sometimes there isn't an obvious way to structure your components and interfaces
  - The challenge then becomes to impose a structure that you think will work
- It can seem tedious
  - …until you've shot yourself in the foot enough times not doing this!

Continuous Integration

- When you refine a component, add it right away to the rest of your code/hardware
- See improvement if it works
- If it doesn't work, you can spot errors by checking the interfaces and localizing the errors right away

Making life easier #1: Using conditional compilation

- Use #ifs or #ifdefs to conditionally enable/disable pieces of your code
  - If you want to try out multiple alternate implementations of a component one after another, and you have code for each, use the same function name with different conditionals
  - You could create a debug version of your code with extra code that just tests your actual code
- Makes it easy to swap out different approaches and versions of your firmware components

Making life easier #2: Using git branches

- Common approach:
  - Keep the main line as the most stable version of the code
  - Create branches as you need based on any scheme you like
    - For each person
    - For each feature that the team happens to be working on
    - Pick one scheme and stick to it
  - Merge into main line after testing
- Using branches also gives you a place to experiment with your code
  - With the assurance that if it doesn't work out, it's one simple command to reverse everything

Making life easier #3: Finite State Machines

- Many complex systems can be modelled as finite state machines
- Think: What do FSMs naturally provide that ties into what we've discussed?
  - Each state should "hide information" from other states – that's why you make transition conditions explicit

Penn Engineering
University of Pennsylvania

Finite State Machines (FSM) in Embedded Systems (Part 1) – Massimiliano Pagani (embeddedrelated.com)

Important Programming Concepts (Even on Embedded Systems) Part V: State Machines - Jason Sachs (embeddedrelated.com)

Big Ideas

- Focus on interfaces first
- Start with simple components
- Integrate early
- Improve incrementally and iteratively