Agile Gets Physical: Slice-Based Integration

Kathleen A. Iberle

kiberle@kiberle.com

Abstract

Are you struggling to integrate the results of your agile software development into a hardware project? Frustrated by waiting weeks to receive test results, or rushing to deliver code only to find the hardware isn't ready to run that code? Do you feel like your hardware partners just don't understand what you're doing?

This paper explores applying agile ideas to a field which isn't entirely agile. We'll show how to use the "integration by slice" method to bring together and test the results of hardware and software development incrementally, planning in a fashion that serves the differing demands of both disciplines.

This paper is adapted from two chapters in my recently published book *When Agile Gets Physical: How to Use Agile Principles to Accelerate Hardware Development*. The book was written for hardware engineers, while this paper takes the perspective of the software tester.

Biography

Kathy Iberle *has been working in Lean and Agile product development for over twenty years, using cutting-edge methods to help strengthen development and make mixed hardware/software projects run more smoothly. Much of Kathy's career was spent at Hewlett-Packard, working on a range of products from printers and electronic instruments to medical applications and website apps. She has a wide variety of experience, having worked in roles from developer to quality assurance expert to agile adoption leader. Since 2012, Kathy has been running her own consulting firm to help companies strengthen and accelerate their product development.*

1 Introduction

When a product incorporates significant amounts of both software and hardware, integrating the work of the various engineering teams often leads to confusion and conflicts. Code is delivered into integration, only to find that the hardware isn't ready yet, and vice versa. Defect reports arrive long after the code was delivered and the software team moved on to other work. Agile firmware and software teams in particular often feel as if their mechanical or electrical engineering partners are operating in a different world and don't understand their agile work.

This paper presents a solution based on the principles behind agile software development. Applying agile software methods verbatim to mixed hardware/software development generally isn't helpful, but many of the tools of agile can be helpful. The key is understanding why the problems are happening, and choosing the tools which address those particular problems.

2 Why is Hardware/Firmware Integration So Hard?

Hardware and firmware development teams usually have very different product development processes, which leads to differing assumptions about the integration process. When firmware teams are using agile, and the hardware teams are using a phase-gate or "waterfall" style of development, their methods often collide with each other, causing frustration on both sides. Yet the two engineering disciplines are using different methods for very good reasons.

2.1 Different Ways to Deal with Uncertainty

Both hardware and software development operate in a realm of significant uncertainty. We can't tell in advance exactly what the customer really wants, or predict the security needs a year in advance of release. We also can't predict exactly how a physical object will work when it is finally built with realistic parts, or precisely what the materials and manufacturing process must be to achieve the desired behavior.

One approach to dealing with uncertainty is to try things on a small scale to see if they work. Agile software development does exactly this. The development is broken into small batches of features, and each small batch is driven to completion in a single sprint or iteration. The feedback loop – discovering and fixing errors or incorrect assumptions – is short and fast. There hasn't been time for the developer to forget the details, there's no need to roll build and test tools back to earlier versions, and the error hasn't been repeated in other related code. Short feedback loops are less expensive.

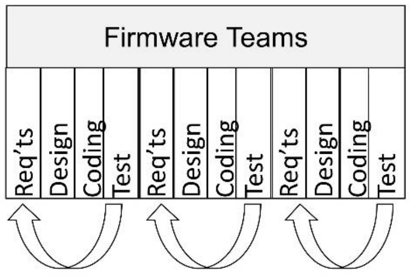


Figure 1: Firmware development using agile has short, fast loopbacks.

"Try it and see" would work in hardware too, except for one thing – it's relatively easy to rework software if your guess is wrong, but it is much more difficult and time-consuming to change hardware. Software, especially software developed with the assumption it may need to be changed, can often rewritten and retested in days or even hours. Revising hardware, particularly mechanical parts, may take weeks, because physical tooling must be revised or recreated. This is known as a *high cost of change*.

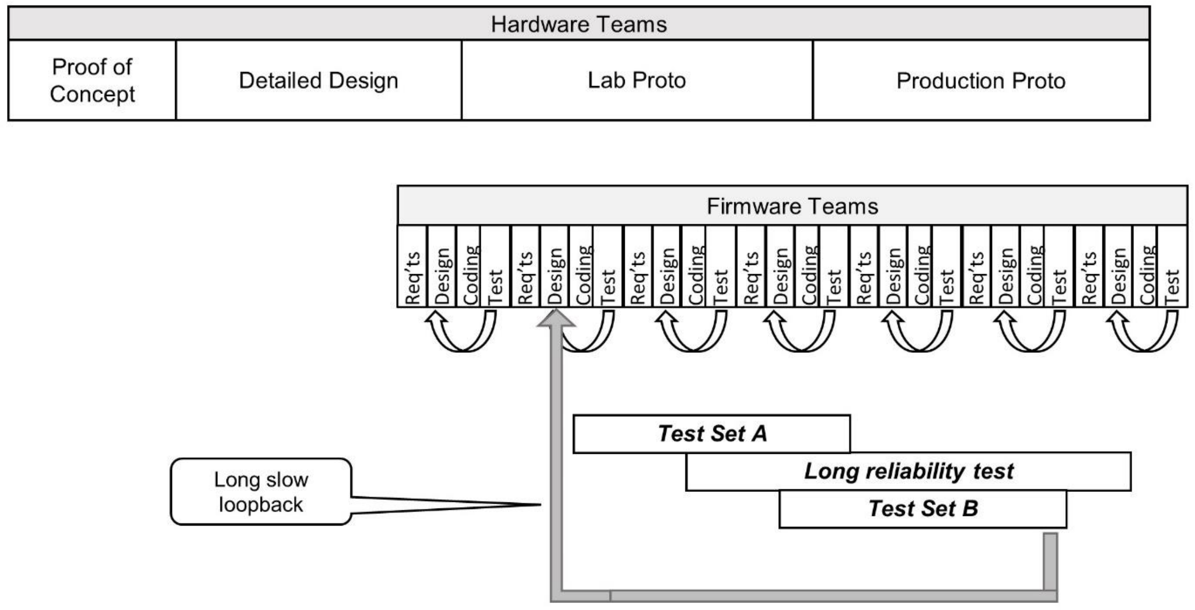
Teams facing a high cost of change want to get it right the first time, so they often use a phase-gate process which looks a lot like waterfall software development. In addition, mechanical features often can't be developed independently of other features, so the "user story" approach isn't entirely feasible. You can't tell how well a handle will work by itself – the rest of the object must be attached. These two factors mean that a hardware team typically develops most features together in one huge batch. The batch progresses through several levels of "realness" from an initial rough mockup through increasingly realistic prototypes to a final manufacturable product. The team also has to concurrently develop the manufacturing process.

These two methods are pretty different, but that typically doesn't cause problems in the early stages of a hardware/firmware project. That's because the engineering disciplines are often working more or less independently of each other at the start. The firmware teams are using software stubs to simulate the actual hardware, or testing with older versions of the product or breadboard emulators[1]. The hardware teams often use the previous product's firmware or a simple throw-away firmware test harness to drive just the behavior they're currently working on. Since the disciplines are largely operating independently, they can use different project management methods without interfering with each other.

2.2 Hardware-Style Integration Creates Long, Expensive Loopbacks

The problems show up when the time comes to integrate the firmware with fairly realistic hardware. Here's where we learn that the emulator or the test harness didn't accurately represent the final component. Learning something new isn't necessarily a problem – but **when** you learn it can be problematic. Unfortunately, traditional hardware-focused integration tends to push the time out, creating long, expensive loopbacks instead of short loopbacks.

Traditional hardware-focused test planning minimizes the need for expensive physical prototypes and test beds by organizing the testing into long test suites which test multiple aspects of the system at once. Often tests are checking functionality and collecting statistical data on behavior at the same time. This reduces the need for costly prototype units, but it breaks the connection between when a feature is delivered and when it's tested. This creates long loopbacks, as shown in Figure 2.



[1] There's a good example of this in chapter 6 of *A Practical Approach to Large-Scale Agile Development* (Gruver et. al. 2013).

Figure 2: Hardware-style integration causes long, slow loopbacks for firmware teams.

2.3 Large Test Suites Lead to Confusion

Hardware-style integration testing usually reports defects well after the code was delivered, often in batches at the end of each test suite. An agile firmware team may respond with several quick deliveries of new firmware – but the hardware-style integration testing has already started another test suite. Their process isn't designed to handle new revisions in the middle of a test suite. The program doesn't have adequate tools to keep track of what is supposed to be working or not working at a given point, and which revisions of firmware require which sets of hardware.

If the product can be driven or accessed from mobile apps or the cloud, that adds another team or two, which are also firing off revisions of their code. By the middle of integration, I often see a barrage of deliveries - updated hardware subsystems, firmware drops, and drops of cloud-based software services – which aren't coordinated with each other or with the test plan. Relationships are unclear, as shown in Figure 4. It's very easy to lose track of what needs to be done in order to deliver a working product.

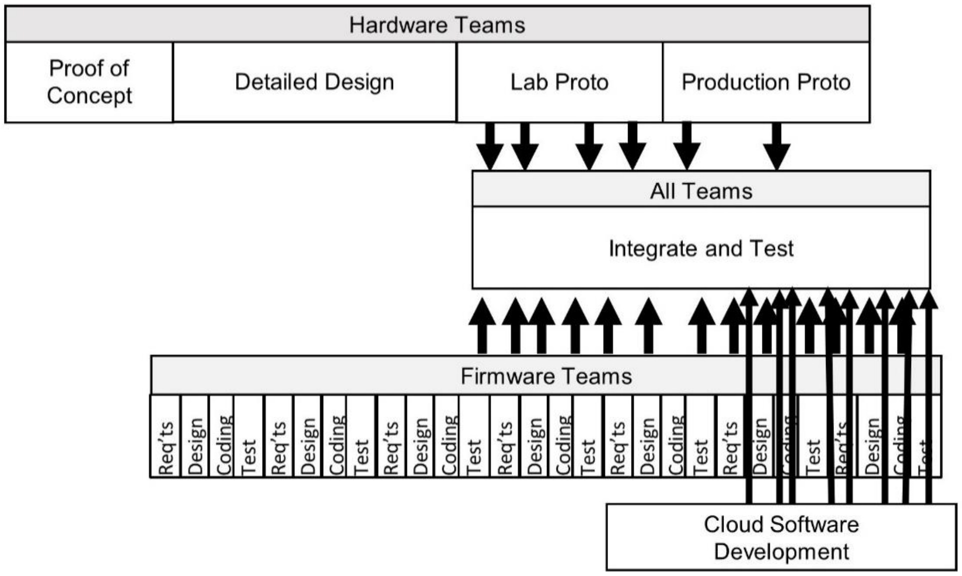


Figure 3: Unclear relationships between software drops, hardware drops and integration tests.

This problem is exacerbated by "fixing it in software". It often makes economic sense to fix a problem with a firmware change rather than reworking the hardware, but that can wreak havoc on firmware project estimates because those fixes are essentially additional features. And it creates yet more need for yet more drops. Chaos can ensue.

Fortunately, applying the principles which underly agile methods results in a much better solution.

3 The Solution: Smaller Batches of Integration

The root cause of this chaos is the mismatch between the large batches of work represented by those long integration test suites and the small batches used by firmware and software development teams. There's no way to set up a clear correspondence, as shown in Figure 4.

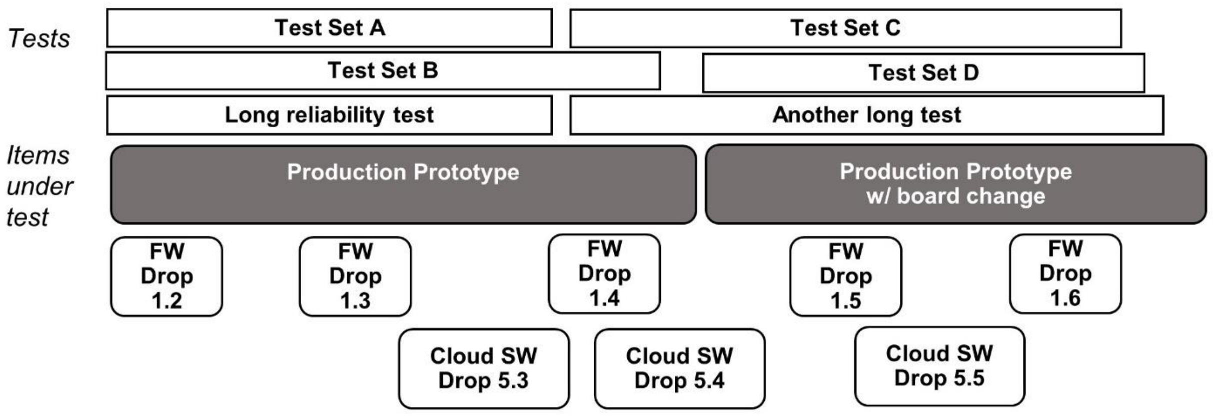


Figure 4: A typical integration plan with long test suites.

The software batches are very small, but we don't want to make them larger. That would create longer loopbacks. The hardware batches are very large, but making them smaller isn't very practical due to the high cost of change and the difficulty of developing a feature independent of the other features.

A third path is to split *the integration testing itself* into smaller batches of tests *and* focus each of those batches on specific user-visible features or capabilities. This gives us shorter batches of testing with shorter feedback loops, and the focus on user-visible features makes it much easier to see the relationship between development drops and test suites.

3.1 Start by Defining Timeboxes to Create a Cadence

I start by splitting the integration schedule into arbitrary, evenly spaced timeboxes of a few weeks' length, as shown in Figure 5. If the firmware team is using agile, the firmware team's sprint or iteration length is often a good choice unless it's shorter than two weeks. A timebox of less than two weeks is rarely optimal for hardware/software integration testing due to the additional overhead.[2] The integration tests have to use real physical objects, so they take longer to set up and tear down than a firmware subsystem test.

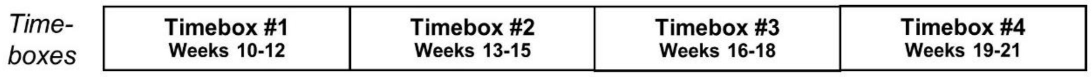


Figure 5: Integration Timeboxes Create a Cadence for Planning

Simply creating these timeboxes sometimes improves the project management immediately. The cadence of the timeboxes simplifies schedule discussions by reducing the degrees of freedom, just like sprints. The teams don't talk about which day to deliver a drop, but rather which timebox. That's often a much simpler decision.

3.2 Split Up the Test Suites into Smaller Batches

There's still a mismatch between the fast firmware deliveries and the long test suites. The test suites need to be split into smaller suites which will more or less fit into a timebox, as shown in Figure 6. Each of these suites will focus on a specific set of features or capabilities. This won't be perfect – there's usually still a few long test suites, such as those collecting statistical data on repetitive actions.

[2] What is an optimal length? See my online article "Cadence: How Fast is Too Fast?" (Iberle 2021)

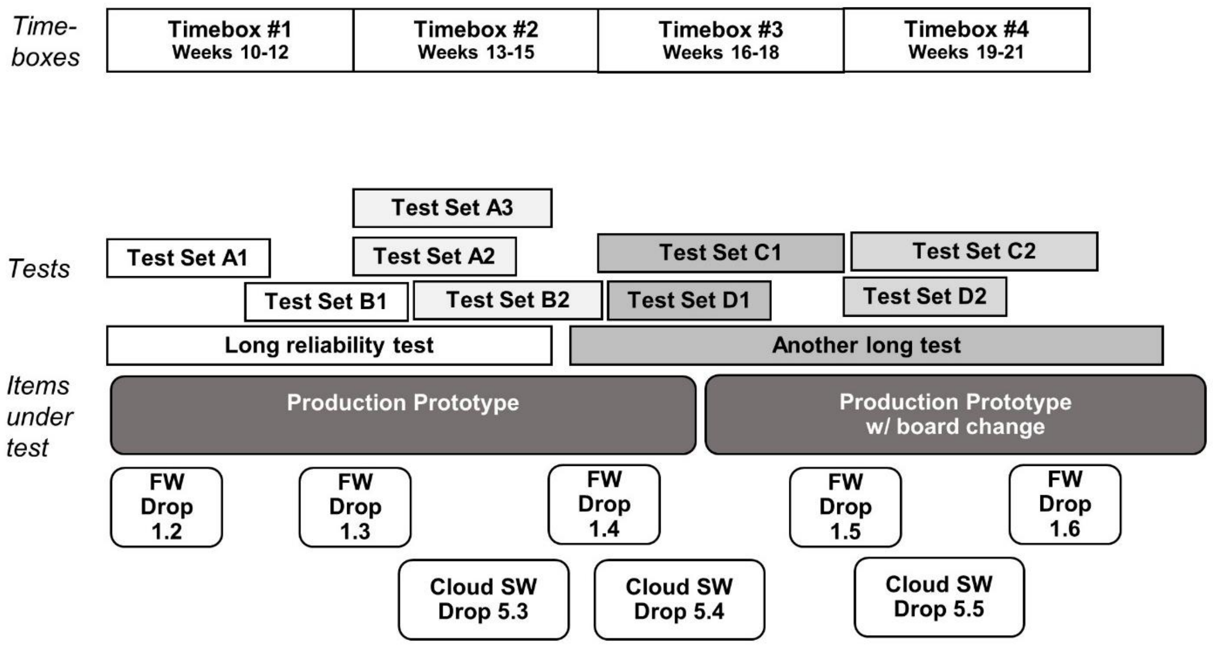


Figure 6: Test Suites are Split Into Smaller Batches.

3.3 Identify the Development Which Is Needed by the Test Suites

Then, for each timebox, the hardware and software teams together identify the features which must be delivered in order for the planned test suites to pass. Those features are then treated as a group, which we call a *slice*. A slice is a set of features or behavior that will be tested within a single integration timebox. A slice is much larger than a typical user story – it's more the size of an epic or even bigger.

The slices pull the disparate development efforts together, so the firmware, hardware, and test teams are all aiming for the same goal at the same time. The slice definitions provide a solid target for each timebox, as shown in Figure 7. It may seem odd to define a group of features by starting with a test rather than product requirements, but this is quite effective in practice.

Slice-based integration is considerably easier to manage than the more traditional method, because there's a clearer relationship between the test suites and the firmware deliveries. When using a slice-based planning method:

- All development teams know in advance what does and doesn't need to be delivered for any given timebox.
- Test teams can determine whether or not those items have been delivered – which tells the program whether or not the system is ready to run (and pass!) a given test suite.
- Teams get faster feedback, because less time passes between writing software / building the prototype hardware and receiving test results from the integrated system.
- Program management can more easily see what is done, and what still needs work.

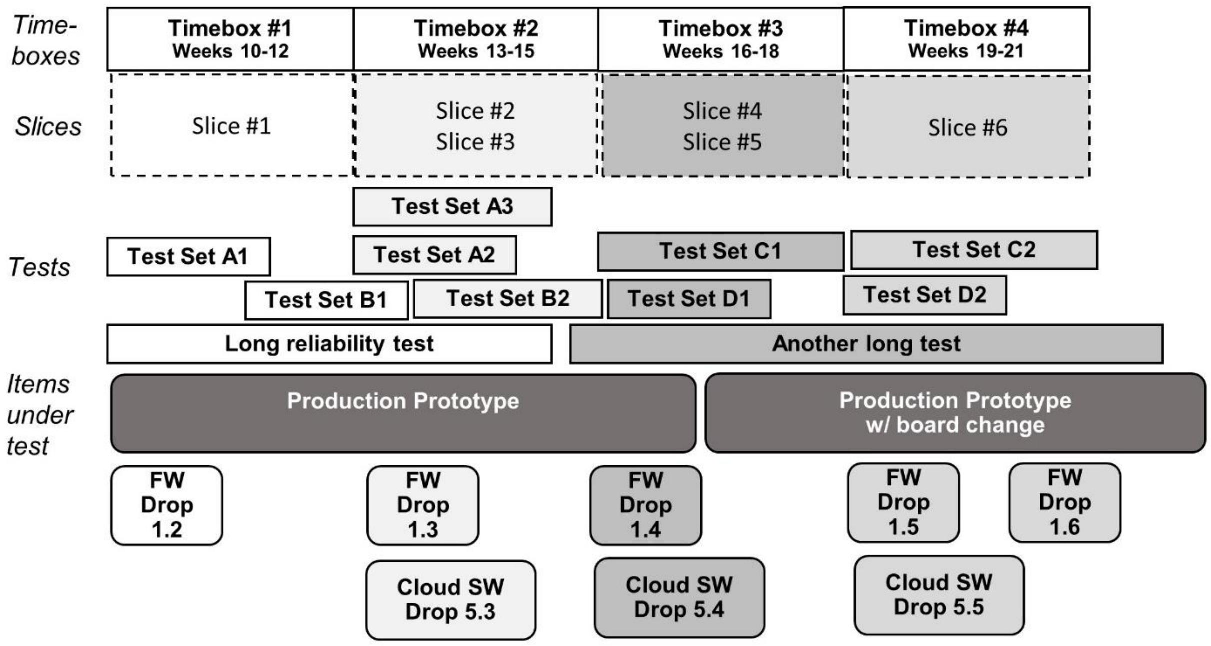


Figure 7: Integration By Slice Coordinates Hardware Prototypes, Firmware Drops, and Tests.

3.4 What's a Good Slice?

Typically, the functionality in a single slice requires contributions from multiple disciplines and subsystems. That's why it is called a slice - each batch of functionality slices through the architecture just like slicing through a layer cake. Bill Wake used this metaphor in an Internet article "INVEST in Good Stories and SMART Tasks", where he described slicing vertically through the architecture layers to provide the customer with "the essence of the whole cake". (Wake 2003) The same principle works for a physical product plus its firmware.

Figure 8 shows some possible slices for a hypothetical inkjet printer:

- Slice #1: Printer can print a test page in response to a button press.
- Slice #2: Printer can print a page sent from a connected PC.
- Slice #3: Printer can print a page sent from a mobile app.
- Slice #4: Printer can remind user to purchase more ink.

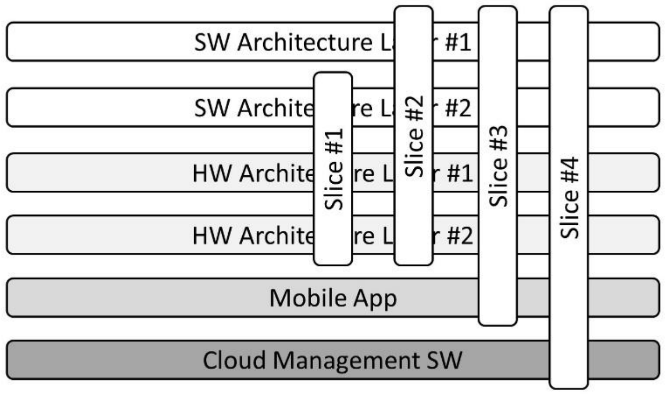


Figure 8: Slices through the architecture of an inkjet printer.

The slices in this example describe high-level end-to-end behaviors from the end user's perspective. Your test plans may refer to these as black-box tests, end-to-end tests, or user scenario tests.

The manufacturing process is also a user, so we have slices capturing behavior seen by the manufacturing process:

- Slice #19: Printer can run the production test which verifies alignment of the cartridge.

- Slice #20: Printer can perform post-production tests which verify that most recent firmware has been loaded.

And lastly, the product will need slices which test the non-functional requirements or capabilities of the system: security, usability, reliability and the like.

- Slice #15: Known paths for hacking into network via mobile app connected to printer are blocked.

A good definition for this type of slice enables the team to understand what must be finished before it is sensible to run these tests. If your organization is prone to running tests too early, this can save a lot of time. Writing the definition also is a good opportunity to clarify expectations for the capability. What does “reliable” actually mean? And how will you know if that expectation is met?

Slices are effective only when all teams understand what each slice means and can relate it to their own work and to the integration testing. Teams need a common language for effective coordination. The user’s language is the one most shared across teams, so that’s what we use. “Confirm torque required to produce adequate sealing for the R3448 Pressure Valve” or “Error-handling for temperature controls” may be meaningful to one team, but they don’t tell the other teams what they need to contribute to this slice. When each slice delivers behavior that is meaningful for the end user, it’s also easier to assess progress across the entire program.

If we must refer to the architecture, the slice should still be stated in terms recognizable by a user, such as “Printer functions normally with Rev 4.2 boards” rather than “run regression tests with Rev 4.2 boards”. This also encourages the test team to ask interesting questions regarding what “functions normally” actually means.

Attributes of a Good Slice

An effective slice, however, has more than just a clear scope. A good slice:

- Has a name and definition which clearly communicate its scope
- Enables test teams to identify or describe the tests which will validate the slice
- Enables development teams to easily identify what to deliver so those tests will pass
- Is small enough to be tested within one timebox
- Delivers a usable set of behaviors, or a capability that some user cares about, or can answer a question critical to further development
- Bonus points if the slice allows the team to test riskier aspects of the system early

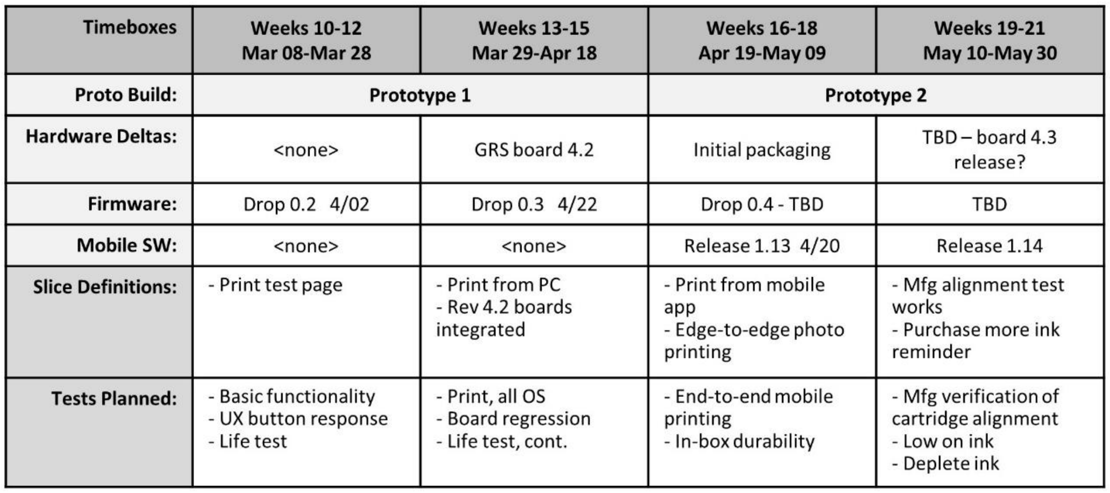
Splitting the integration work into slices makes it much easier to coordinate the various teams to deliver the right functionality all at once time. And the integration batches are smaller, which creates faster feedback and reduces those long, slow loopbacks.

3.5 The Slice-Based Integration Plan

A slice-based integration plan has four main parts, as shown in Figure 9.

- Timeboxes: The weeks remaining in development, organized into evenly-sized boxes. Typically, these timeboxes are two to four weeks long, and smaller is better. If possible, align with the firmware team’s sprints or iterations so that the firmware team drops new software into integration at the start of each timebox.
- Items Under Test: the deliverables that are in testing during a timebox. These comprise:

- hardware prototypes or prototype variants that will be used for the testing
    - firmware drops
    - cloud software drops
- Slices: the *features and behaviors* to be delivered in this timebox. The items under test have to come together to deliver these features.
- Tests: The specific tests planned for this timebox which will exercise the delivered features.



Timeboxes	**Weeks 10-12 Mar 08-Mar 28**	**Weeks 13-15 Mar 29-Apr 18**	**Weeks 16-18 Apr 19-May 09**	**Weeks 19-21 May 10-May 30**
Proto Build:	**Prototype 1**		**Prototype 2**	
Hardware Deltas:	<none>	GRS board 4.2	Initial packaging	TBD – board 4.3 release?
Firmware:	Drop 0.2 4/02	Drop 0.3 4/22	Drop 0.4 - TBD	TBD
Mobile SW:	<none>	<none>	Release 1.13 4/20	Release 1.14
Slice Definitions:	- Print test page	- Print from PC - Rev 4.2 boards integrated	- Print from mobile app - Edge-to-edge photo printing	- Mfg alignment test works - Purchase more ink reminder
Tests Planned:	- Basic functionality - UX button response - Life test	- Print, all OS - Board regression - Life test, cont.	- End-to-end mobile printing - In-box durability	- Mfg verification of cartridge alignment - Low on ink - Deplete ink

Figure 9: Anatomy of a Slice-Based Integration Plan.

A slice-based integration plan looks similar to an agile roadmap[3], and is managed similarly with rolling-wave planning. Just as in agile development, the team meets at least once per timebox to review the plan and adjust it based on recent learnings. This plan can be kept in a spreadsheet, or an HTML table, or a physical planning board. The ideal tool is visible to all your team members and relatively easy to change – because it will be changed frequently.

3.6 Build a Slice-Based Integration Plan

Moving from a traditional hardware-centric integration plan to a slice-based integration plan can be awkward, since it's rarely practical to re-plan all the integration testing from scratch. Here's one way to create a slice-based integration plan, starting from where you are today. This is best done with all the technical leads from development and testing teams in the room together.

Starting from your existing plan:

1. Define your timeboxes.
2. Place the known plans for hardware prototypes into the "Items In Test" row.
3. Add the planned firmware and cloud software drops, using your usual naming conventions. At this point, some of the drops will have intended contents and others may not yet be planned. That's OK.
4. Draw the planned integration tests on your plan and summarize the functionality each one assumes.
5. Create the first draft of your slices. In the "Slice Definition" row, summarize the planned contents of the firmware and hardware drops – *stated in the form of end-to-end testable behavior*. This

[3] There's a good explanation of an agile roadmaps and rolling-wave planning in chapter 6 of Johanna Rothman's *Create Your Successful Agile Project*. (Rothman 2017)

language will allow you to compare the test plan with the delivery plan. This is usually the epic level, not the user story level.

At this point, there's probably a bunch of mismatches between the planned testing and the planned delivery, and a lot of things that are ambiguous or blank.

Start iterating towards a better plan by comparing the slices with the planned integration tests. If the slices are delivered as shown, can the integration tests be run successfully? If the test plan calls for testing something that won't be ready until a later timebox, there are three options:

1. Move the test later.
2. Move the feature development earlier. This may mean reorganizing some firmware drops.
3. Split the test into smaller pieces and put them in different timeboxes, so we can test what's ready as soon as it is ready (but not sooner).

Keep iterating on your plan until you have a schedule that looks feasible given what you know today.

When you have a plan which aligns all the disciplines with a clear set of targets at a manageable level of detail, you're done. The teams now have a shared view of the program's path towards full integration that accounts for the cross-discipline dependencies. Once they have that view, they won't need a detailed Gantt chart showing all these dependencies in great detail. Instead, the technical leaders can use their slice-based integration plan to adapt to changing conditions and manage the complexity by working together. The timeboxes provide a regular cadence for this coordination, and small enough batch sizes to shorten those feedback loops and make progress visible enough to be actionable.

Don't try for perfection. If you have a few tests which span more than one timebox, that's ok. And don't feel you have to plan the entire integration plan all the way through the program. As in agile development, it is ok to have some iterations without defined contents or only roughly defined contents. You'll be updating the plan regularly and frequently.

3.7 Alternate Methods

Sometimes it makes more sense to start from the test suites rather than from the firmware drops. If the test suites already are organized around user-visible behavior, it may be easier to pull out those behavior statements and use them to create the first draft of slices. A single test suite generally tests a good-sized list of behaviors which will have to be split into different slices. The development and test teams will work together to decide which behaviors go in the first slice and which in the next slice, and so forth.

It's also possible to do both. Start with the firmware drop definitions (usually at the epic level) and work towards a set of slices which are smaller than the original test suites and larger than the firmware epics. When the end-to-end behavior isn't clear from the firmware drop definitions, look for that type of behavior in the tests.

It might seem sensible to start defining slices by splitting up the requirements list, but in practice I haven't seen this work well. For one thing, the requirements specification often has no relationship with the order in which features will be developed, whereas the development plans and test plans do. Plus, it's not unusual to find that many requirements aren't stated in the requirements specification at all (although they may be documented in test cases).

3.8 Use Rolling Wave Planning to Update the Integration Plan

Your team leaders will meet at least once per timebox to review the slice-based integration plan and adjust it based on recent learnings. These meetings should be frequent, short, and on a regular schedule. For many teams starting out, once per timebox is not quite often enough. Once a week may be better.

These planning meetings bring together the technical leads and project leaders from all of the disciplines, including the system or integration test groups. Each discipline needs at least one person who can share progress updates and make decisions. The meeting might be run by the overall project leader, or by a chief engineer or architect, or even the integration test lead.

The typical agenda is:

1. Review current work and make adjustments as needed.
2. Review the upcoming timebox and make adjustments as needed.
3. Fill in the next open timebox with a fully detailed plan.

We don’t use the traditional Scrum standup questions because those are designed for load-balancing the work across the team members. In most mixed hardware/software programs, it’s not practical to shift work across disciplines, so that’s not our goal. Instead, our questions are designed to share information efficiently:

1. Has any item-in-test or planned testing changed?
2. Is anything in your way of getting ready for the next timebox?
3. What else do we need to know?

The purpose of the meeting is to adjust the integration plan based on the most current information. It’s not a progress report, so the only thing displayed is the integration plan. There’s no need to cover things which have already been communicated in another meeting. This meeting is a place to make sure we all understand what to do next, and to identify what might get in our way.

As in any technical meeting, there is a temptation to start solving knotty problems in the meeting. Don’t do that. Unless it can be solved in a few minutes – “Does anyone here know where the extra 4.2 boards are?” – the issue should be raised and then taken off-line.

3.9 This Is Not a Game of Perfect

The primary benefit of a slice-based integration plan is better visibility and communication across teams. Each team can see where the program is going, and what they need to do individually and collectively to get there.

A healthy integration plan has these characteristics:

- The timeboxes are roughly equal to the size of firmware sprints / iterations.
- The slice definitions describe recognizable, testable functionality.
- The tests align well with the slices.
- It’s obvious what we expect to learn from each test suite.
- The plan is visible, and updated regularly with all disciplines actively participating.
- The participants feel that the meetings are worth their time.

You’ll know your slice-based planning method is working when you can change the plan without causing much rework for any of the disciplines. And you’ll know it’s working when you’re spending less time feeling confused or fixing problems caused by miscommunication. People know what they need to deliver when, and what to do if their plans need to change.

If the only benefit from slice-based integration testing is that hardware, software and test teams stop arguing so much and work more effectively together, that can be a big win all by itself. But you can go further.

4 Towards a More Agile Approach

4.1 Risk-Based Prioritization

Once the slice-based integration plan has provided an explicit correspondence between firmware and hardware deliveries and the tests, and the test suites are smaller, it's easier to move the more critical tests earlier. The "more critical" tests are those which are likely to find either a lot of defects or defects which are particularly time-consuming to fix.

This discussion can lead to another useful discussion: whether some **development** should also be shifted earlier to allow early testing. That might be defect-prone areas, or it might be areas where learning a few things early would provide useful guidance for later development.

You might also see opportunities to shift left – move some integration testing back into subsystem testing. When an integration test finds numerous defects that could have been found without testing on a fully integrated system, some coverage should be moved back into subsystem testing. Again, we will shorten a feedback loop.

4.2 Smaller, Earlier Slices

In agile development, we know that smaller batches of features give us feedback sooner, and that's an advantage. The same thing is true in integration testing. .

Smaller slices deliver value to the program earlier. An integration slice with its testing delivers value by providing information about the quality and acceptability of those particular features *and* about the progress of the project. When teams know that they have *finished* features to an acceptable level of quality and acceptability, they can use that knowledge to measure progress with certainty.

We can accelerate even more by starting integration by slice earlier in development. Slices nearly always consist of working features because working features deliver the most information about the usefulness and quality of the planned features and about the progress of the project. (Sound familiar?)

However, the earliest slices may not be all that compelling from a user perspective. The earliest possible slices often resemble the "walking skeletons" that authors like Johanna Rothman describe as the earliest outcome of Agile Software Development. Early slices might even include some that aren't user-focused at all, but instead test whether major subsystems are playing nicely together. For instance, a system controlling multiple objects in real-time might have a slice which demonstrates that the communications are in place and happening fast enough.

4.3 Emergent Features

Some behavior can only be approximated until the physical system is quite realistic. When behavior depends on precise control of a physical object, the exact size and properties of the parts will affect the behavior, as shown in Figure 6.3.

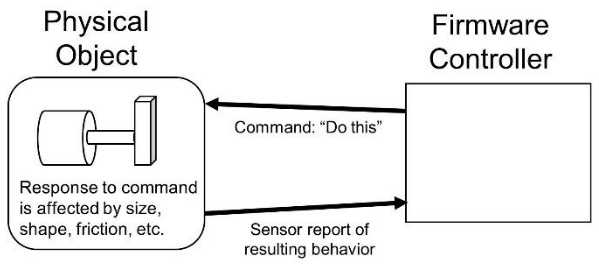

Physical Object
Firmware Controller
Command: "Do this"
Response to command is affected by size, shape, friction, etc.
Sensor report of resulting behavior

Figure 6.3: Emergent Behavior

This firmware controller must be developed in stages, since the final behavior emerges gradually as successive hardware prototypes become more and more realistic. I call this "emergent behavior". If a product requires precision control of mechanical or electrical parts or involves fluid dynamics, chances are that it has some emergent behavior. The right firmware settings for this behavior can't be predicted from the design nor determined by early prototypes, but rather must be found by trial and error using the final production prototypes.

Color printing in inkjet printers involves emergent behavior. Printing in color requires precise alignment of multiple ink colors. The ink cartridge, motors, controllers and firmware have to work together to control the movement of the paper and the firing of the cartridge nozzles at the same time. The firmware settings which provide this control for early prototypes likely will not be the settings needed for the final product. An early prototype, or even the final lab prototype, doesn't use the same manufacturing process or sometimes even the same materials as a production printer. These differences make the parts of the overall system move slightly differently, and so adjustments will need to be made. Firmware teams know this and design their code with parameters they can update to quickly dial in the right values once the final hardware is available. Program management sometimes isn't aware this is happening.

When there's emergent behavior, our slices are designed to provide feedback early and shorten those feedback loops, without assuming we are done when we know we are not. For instance:

- Slice #11: Nozzle and roller positioning is accurately controlled by configurable firmware parameters across the full range of the parameters.

- Slice #12: Printer prints in color with correct colors and alignment which appears correct without magnification.

- Slice #13: Printer prints in color with precisely correct color registration using production prototype printers and production ink cartridges.

Slice #11 obviously isn't an end-to-end user-visible behavior (unless you consider the firmware developer to be a user) but it is a very helpful slice. This tells the rest of the team that the firmware team is able to tweak the parameters later on and get the desired results. It's easier to get this nailed down early, so the fine-tuning at the end won't trip over a bug in parameter handling.

Setting up a series of slices like this makes the emergent behavior much more visible to the program team. It becomes obvious why some of the firmware needs to freeze later than the hardware, and where in the schedule firmware teams and test teams have budgeted time for finalizing that emergent behavior.

4.4 Other Opportunities

Many readers will observe that there are other opportunities in a hardware project to take a more agile approach. These opportunities are further discussed in *When Agile Gets Physical*. In short, breaking the work into user stories doesn't really work, but it is possible to break the work into other types of batches which can be completed in timeboxes and deliver immediate value. Namely,

- In early development, we split **the investigative work and research** into batches. Each batch delivers the knowledge to make an important product decision accurately.
- In middle development, we split **routine development work into batches.** These batches don't deliver finished features, but rather deliverables which answer more key questions about the product and feed into the next step of development.
- In late development, we split **the integration testing** into smaller batches. Each batch delivers specific information about the quality of a set of features or attributes of the product,

If you want to read more about why other types of batches can work (and when they don't), see my series of blogs on this topic https://kiberle.com/2022/04/05/agile-for-hardware/.

5 Conclusion

Slice-based integration improves the traditional hardware/software integration plan by using familiar agile principles.

- Small batches of work
- Batches which deliver user-visible behavior or features
- Cadence (in the form of timeboxes) to simplify planning
- Rolling-wave planning: adjust to reality as you go

Application of these principles in a way that takes into account the different constraints of hardware development and software development provides a method that can substantially simplify the coordination between different engineering disciplines and speed up the program.

Slice-based integration works well because it gives all the teams a common planning method and language which is meaningful to everyone, and it gives the teams opportunity to shorten the loopbacks and save time on the program.

You can learn more about slice-based integration testing and the more advanced "integration train" method in our new book *When Agile Gets Physical: How to Use Agile Principles to Accelerate Hardware Development* by Katherine Radeka and Kathy Iberle. (Radeka 2022)

References

Gruver, Gary. Young, Mike. Fulghum, Pat. 2013. *A Practical Approach to Large-Scale Agile Development: How HP Transformed LaserJet FutureSmart Firmware*. Addison-Wesley.

Iberle, Kathleen. "Cadence: How Fast is Too Fast?". https://kiberle.com/2021/02/14/cadence-how-fast-is-too-fast/ Posted February 14, 2021. Downloaded August 20, 2022.

Iberle, Kathleen. "Agile for Hardware". https://kiberle.com/2022/06/10/what-makes-agile-work-2/. Posted February 21, 2022. Downloaded August 20, 2022.

Iberle, Kathleen. "What Makes Agile Software Development Work?". https://kiberle.com/2022/04/05/agile-for-hardware/. Posted February 10, 2020. Downloaded August 20, 2022.

Rothman, Johanna. 2017. *Create Your Successful Agile Project: Collaborate, Measure, Estimate, Deliver*. Raleigh, North Carolina: Pragmatic Bookshelf.

Rothman, Johanna. 2016. *Agile and Lean Program Management: Scaling Collaboration Across the Organization*. Arlington, Massachusetts: Practical Ink.

Radeka, Katherine and Iberle, Kathy. 2022. *When Agile Gets Physical: How to Use Agile Principles to Accelerate Hardware Development*. Camas, Washington: Chesapeake Research Press.

Wake, Bill. "INVEST in Good Stories, and SMART Tasks". https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/). Posted August 17, 2003. Downloaded November 15, 2021.