Bi-weekly Random Bits from the Internet

2015-10-17

(TOO MUCH ALL AMERICA FAST FOOD OVER HUNTINGTON BEACH)

Machine Learning: The High-Interest Credit Card of Technical Debt

P2, D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary and Michael Young, NIPS Workshop 2014

Why It's Often Easier To Innovate In China Than In The United States

P15, Andrew 'bunnie' Huang, Gizmodo, June 15 2015

Origins and Mission of the Federal Reserve

P38, Ben Bernanke, The Federal Reserve and the Financial Crisis, March 2012

Machine Learning: The High-Interest Credit Card of Technical Debt

D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary and Michael Young, NIPS Workshop 2014

1 Machine Learning and Complex Systems

Real world software engineers are often faced with the challenge of moving quickly to ship new products or services, which can lead to a dilemma between speed of execution and quality of engineering. The concept of technical debt was first introduced by Ward Cunningham in 1992 as a way to help quantify the cost of such decisions. Like incurring fiscal debt, there are often sound strategic reasons to take on technical debt. Not all debt is necessarily bad, but technical debt does tend to compound. Deferring the work to pay it off results in increasing costs, system brittleness, and reduced rates of innovation.

Traditional methods of paying off technical debt include refactoring, increasing coverage of unit tests, deleting dead code, reducing dependencies, tightening APIs, and improving documentation [4]. The goal of these activities is not to add new functionality, but to make it easier to add future improvements, be cheaper to maintain, and reduce the likelihood of bugs.

One of the basic arguments in this paper is that machine learning packages have all the basic code complexity issues as normal code, but also have a larger system-level complexity that can create hidden debt. Thus, refactoring these libraries, adding better unit tests, and associated activity is time well spent but does not necessarily address debt at a systems level.

In this paper, we focus on the system-level interaction between machine learning code and larger systems as an area where hidden technical debt may rapidly accumulate. At a system-level, a machine learning model may subtly erode abstraction boundaries. It may be tempting to re-use input signals in ways that create unintended tight coupling of otherwise disjoint systems. Machine learning packages may often be treated as black boxes, resulting in large masses of “glue code” or calibration layers that can lock in assumptions. Changes in the external world may make models or input signals change behavior in unintended ways, ratcheting up maintenance cost and the burden of any debt. Even monitoring that the system as a whole is operating as intended may be difficult without careful design.

Indeed, a remarkable portion of real-world “machine learning” work is devoted to tackling issues of this form. Paying down technical debt may initially appear less glamorous than research results usually reported in academic ML conferences. But it is critical for long-term system health and enables algorithmic advances and other cutting-edge improvements.

2 Complex Models Erode Boundaries

Traditional software engineering practice has shown that strong abstraction boundaries using encapsulation and modular design help create maintainable code in which it is easy to make isolated changes and improvements. Strict abstraction boundaries help express the invariants and logical consistency of the information inputs and outputs from an given component [4].

Unfortunately, it is difficult to enforce strict abstraction boundaries for machine learning systems by requiring these systems to adhere to specific intended behavior. Indeed, arguably the most important reason for using a machine learning system is precisely that the desired behavior cannot be effectively implemented in software logic without dependency on external data. There is little way to separate abstract behavioral invariants from quirks of data. The resulting erosion of boundaries can cause significant increases in technical debt. In this section we look at several issues of this form.

2.1 Entanglement

From a high level perspective, a machine learning package is a tool for mixing data sources together. That is, machine learning models are machines for creating entanglement and making the isolation of improvements effectively impossible.

To make this concrete, imagine we have a system that uses features x1,...xn in a model. If we change the input distribution of values in x1 , the importance, weights, or use of the remaining n − 1 features may all change—this is true whether the model is retrained fully in a batch style or allowed to adapt in an online fashion. Adding a new feature xn+1 can cause similar changes, as can removing any feature xj . No inputs are ever really independent. We refer to this here as the CACE principle: Changing Anything Changes Everything.

The net result of such changes is that prediction behavior may alter, either subtly or dramatically, on various slices of the distribution. The same principle applies to hyper-parameters. Changes in regularization strength, learning settings, sampling

methods in training, convergence thresholds, and essentially every other possible tweak can have similarly wide ranging effects.

One possible mitigation strategy is to isolate models and serve ensembles. This approach is useful in situations such as [8], in which sub-problems decompose naturally, or in which the cost of maintaining separate models is outweighed by the benefits of enforced modularity. However, in many large-scale settings such a strategy may prove unscalable. And within a given model, the issues of entanglement may still be present.

A second possible mitigation strategy is to develop methods of gaining deep insights into the behavior of model predictions. One such method was proposed in [6], in which a high-dimensional visualization tool was used to allow researchers to quickly see effects across many dimensions and slicings. Metrics that operate on a slice-by-slice basis may also be extremely useful.

A third possibility is to attempt to use more sophisticated regularization methods to enforce that any changes in prediction performance carry a cost in the objective function used in training [5]. Like any other regularization approach, this kind of approach can be useful but is far from a guarantee and may add more debt via increased system complexity than is reduced via decreased entanglement.

The above mitigation strategies may help, but this issue of entanglement is in some sense innate to machine learning, regardless of the particular learning algorithm being used. In practice, this all too often means that shipping the first version of a machine learning system is easy, but that making subsequent improvements is unexpectedly difficult. This consideration should be weighed carefully against deadline pressures for version 1.0 of any ML system.

2.2 Hidden Feedback Loops

Another worry for real-world systems lies in hidden feedback loops. Systems that learn from world behavior are clearly intended to be part of a feedback loop. For example, a system for predicting the click through rate (CTR) of news headlines on a website likely relies on user clicks as training labels, which in turn depend on previous predictions from the model. This leads to issues in analyzing system performance, but these are the obvious kinds of statistical challenges that machine learning researchers may find natural to investigate [2].

As an example of a hidden loop, now imagine that one of the input features used in this CTR model is a feature xweek that reports how many news headlines the given

user has clicked on in the past week. If the CTR model is improved, it is likely that all users are given better recommendations and many users will click on more headlines. However, the result of this effect may not fully surface for at least a week, as the xweek feature adjusts. Furthermore, if the model is updated on the new data, either in batch mode at a later time or in streaming fashion with online updates, the model may later adjust its opinion of the xweek feature in response. In such a setting, the system will slowly change behavior, potentially over a time scale much longer than a week. Gradual changes not visible in quick experiments make analyzing the effect of proposed changes extremely difficult, and add cost to even simple improvements.

We recommend looking carefully for hidden feedback loops and removing them whenever feasible.

2.3 Undeclared Consumers

Oftentimes, a prediction from a machine learning model A is made accessible to a wide variety of systems, either at runtime or by writing to logs that may later be consumed by other systems. In more classical software engineering, these issues are referred to as visibility debt [7]. Without access controls, it is possible for some of these consumers to be undeclared consumers, consuming the output of a given prediction model as an input to another component of the system. Undeclared consumers are expensive at best and dangerous at worst.

The expense of undeclared consumers is drawn from the sudden tight coupling of model A to other parts of the stack. Changes to A will very likely impact these other parts, sometimes in ways that are unintended, poorly understood, or detrimental. In practice, this has the effect of making it difficult and expensive to make any changes to A at all.

The danger of undeclared consumers is that they may introduce additional hidden feedback loops. Imagine in our news headline CTR prediction system that there is another component of the system in charge of "intelligently" determining the size of the font used for the headline. If this font-size module starts consuming CTR as an input signal, and font-size has an effect on user propensity to click, then the inclusion of CTR in font-size adds a new hidden feedback loop. It's easy to imagine a case where such a system would gradually and endlessly increase the size of all headlines.

Undeclared consumers may be difficult to detect unless the system is specifically designed to guard against this case. In the absence of barriers, engineers may nat-

urally grab for the most convenient signal, especially when there are deadline pressures.

3 Data Dependencies Cost More than Code Dependencies

In [7], dependency debt is noted as a key contributor to code complexity and technical debt in classical software engineering settings. We argue here that data dependencies in machine learning systems carry a similar capacity for building debt. Furthermore, while code dependencies can be relatively easy to identify via static analysis, linkage graphs, and the like, it is far less common that data dependencies have similar analysis tools. Thus, it can be inappropriately easy to build large data-dependency chains that can be difficult to untangle.

3.1 Unstable Data Dependencies

To move quickly, it is often convenient to consume signals as input features that are produced by other systems. However, some input signals are unstable, meaning that they qualitatively changebehavior over time. This can happen implicitly, when the input signal comes from another machine learning model itself that updates over time, or a data-dependent lookup table, such as for computing TF/IDF scores or semantic mappings. It can also happen explicitly, when the engineering ownership of the input signal is separate from the engineering ownership of the model that consumes it. In such cases, changes and improvements to the input signal may be regularly rolled out, without regard for how the machine learning system may be affected. As noted above in the CACE principle, "improvements" to input signals may have arbitrary, sometimes deleterious, effects that are costly to diagnose and address.

One common mitigation strategy for unstable data dependencies is to create a versioned copy of a given signal. For example, rather than allowing a semantic mapping of words to topic clusters to change over time, it might be reasonable to create a frozen version of this mapping and use it until such a time as an updated version has been fully vetted. Versioning carries its own costs, however, such as potential staleness. And the requirement to maintain multiple versions of the same signal over time is a contributor to technical debt in its own right.

3.2 Underutilized Data Dependencies

In code, underutilized dependencies are packages that are mostly unneeded [7]. Similarly, underutilized data dependencies include input features or signals that provide little incremental value in terms of accuracy. Underutilized dependencies

are costly, since they make the system unnecessarily vulnerable to changes.

Underutilized dependencies can creep into a machine learning model in several ways.

- Legacy Features. The most common is that a feature F is included in a model early in its development. As time goes on, other features are added that make F mostly or entirely redundant, but this is not detected.

- Bundled Features. Sometimes, a group of features is evaluated and found to be beneficial. Because of deadline pressures or similar effects, all the features in the bundle are added to the model together. This form of process can hide features that add little or no value.

- ε-Features. As machine learning researchers, it is satisfying to improve model accuracy. It can be tempting to add a new feature to a model that improves accuracy, even when the accuracy gain is very small or when the complexity overhead might be high.

In all these cases, features could be removed from the model with little or no loss in accuracy. But because they are still present, the model will likely assign them some weight, and the system is therefore vulnerable, sometimes catastrophically so, to changes in these unnecessary features.

As an example, suppose that after a team merger, to ease the transition from an old product numbering scheme to new product numbers, both schemes are left in the system as features. New products get only a new number, but old products may have both. The machine learning algorithm knows of no reason to reduce its reliance on the old numbers. A year later, someone acting with good intent cleans up the code that stops populating the database with the old numbers. This change goes undetected by regression tests because no one else is using them any more. This will not be a good day for the maintainers of the machine learning system.

A common mitigation strategy for under-utilized dependencies is to regularly evaluate the effect of removing individual features from a given model and act on this information whenever possible. More broadly, it may be important to build cultural awareness about the long-term benefit of underutilized dependency cleanup.

3.3 Static Analysis of Data Dependencies

One of the key issues in data dependency debt is the difficulty of performing static

analysis. While compilers and build systems typically provide such functionality for code, data dependencies may require additional tooling to track. Without this, it can be difficult to manually track the use of data in a system. On teams with many engineers, or if there are multiple interacting teams, not everyone knows the status of every single feature, and it can be difficult for any individual human to know every last place where the feature was used. For example, suppose that the version of a dictionary must be changed; in a large company, it may not be easy even to find all the consumers of the dictionary. Or suppose that for efficiency a particular signal will no longer be computed; are all former consumers of the signal done with it? Even if there are no references to it in the current version of the codebase, are there still production instances with older binaries that use it? Making changes safely can be difficult without automatic tooling.

A remarkably useful automated feature management tool was described in [6], which enables data sources and features to be annotated. Automated checks can then be run to ensure that all dependencies have the appropriate annotations, and dependency trees can be fully resolved. Since its adoption, this approach has regularly allowed a team at Google to safely delete thousands of lines of feature-related code per quarter, and has made verification of versions and other issues automatic. The system has on many occasions prevented accidental use of deprecated or broken features in new models.

3.4 Correction Cascades

There are often situations in which model a for problem A exists, but a solution for a slightly different problem A` is required. In this case, it can be tempting to learn a model a`(a) that takes a as input and learns a small correction. This can appear to be a fast, low-cost win, as the correction model is likely very small and can often be done by a completely independent team. It is easy and quick to create a first version.

However, this correction model has created a system dependency on a, making it significantly more expensive to analyze improvements to that model in the future. Things get even worse if correction models are cascaded, with a model for problem A`` learned on top of a`, and so on. This can easily happen for closely related problems, such as calibrating outputs to slightly different test distributions. It is not at all unlikely that a correction cascade will create a situation where improving the accuracy of a actually leads to system-level detriments. Additionally, such systems may create deadlock, where the coupled ML system is in a poor local optimum, and no component model may be individually improved. At this point, the independent development that was initially attractive now becomes a large barrier to progress.

A mitigation strategy is to augment a to learn the corrections directly within the same model by adding features that help the model distinguish among the various use-cases. At test time, the model may be queried with the appropriate features for the appropriate test distributions. This is not a free solution—the solutions for the various related problems remain coupled via CACE, but it may be easier to make updates and evaluate their impact.

4 System-level Spaghetti

It is unfortunately common for systems that incorporate machine learning methods to end up with high-debt design patterns. In this section, we examine several system-design anti-patterns [3] that can surface in machine learning systems and which should be avoided or refactored where possible.

4.1 Glue Code

Machine learning researchers tend to develop general purpose solutions as self-contained packages. A wide variety of these are available as open-source packages at places like mloss.org, or from in-house code, proprietary packages, and cloud-based platforms. Using self-contained solutions often results in a glue code system design pattern, in which a massive amount of supporting code is written to get data into and out of general-purpose packages.

This glue code design pattern can be costly in the long term, as it tends to freeze a system to the peculiarities of a specific package. General purpose solutions often have different design goals: they seek to provide one learning system to solve many problems, but many practical software systems are highly engineered to apply to one large-scale problem, for which many experimental solutions are sought. While generic systems might make it possible to interchange optimization algorithms, it is quite often refactoring of the construction of the problem space which yields the most benefit to mature systems. The glue code pattern implicitly embeds this construction in supporting code instead of in principally designed components. As a result, the glue code pattern often makes experimentation with other machine learning approaches prohibitively expensive, resulting in an ongoing tax on innovation.

Glue code can be reduced by choosing to re-implement specific algorithms within the broader system architecture. At first, this may seem like a high cost to pay—re-implementing a machine learning package in C++ or Java that is already available in R or matlab, for example, may appear to be a waste of effort. But the resulting system may require dramatically less glue code to integrate in the overall system, be easier to test, be easier to maintain, and be better designed to allow alter-

nate approaches to be plugged in and empirically tested. Problem-specific machine learning code can also be tweaked with problem-specific knowledge that is hard to support in general packages.

It may be surprising to the academic community to know that only a tiny fraction of the code in many machine learning systems is actually doing "machine learning". When we recognize that a mature system might end up being (at most) 5% machine learning code and (at least) 95% glue code, reimplementation rather than reuse of a clumsy API looks like a much better strategy.

4.2 Pipeline Jungles

As a special case of glue code, pipeline jungles often appear in data preparation. These can evolve organically, as new signals are identified and new information sources added. Without care, the resulting system for preparing data in an ML-friendly format may become a jungle of scrapes, joins, and sampling steps, often with intermediate files output. Managing these pipelines, detecting errors and recovering from failures are all difficult and costly [1]. Testing such pipelines often requires expensive end-to-end integration tests. All of this adds to technical debt of a system and makes further innovation more costly.

Pipeline jungles can only be avoided by thinking holistically about data collection and feature extraction. The clean-slate approach of scrapping a pipeline jungle and redesigning from the ground up is indeed a major investment of engineering effort, but one that can dramatically reduce ongoing costs and speed further innovation.

It's worth noting that glue code and pipeline jungles are symptomatic of integration issues that may have a root cause in overly separated "research" and "engineering" roles. When machine learning packages are developed in an ivory-tower setting, the resulting packages may appear to be more like black boxes to the teams that actually employ them in practice. At Google, a hybrid research approach where engineers and researchers are embedded together on the same teams (and indeed, are often the same people) has helped reduce this source of friction significantly [10]. But even when a fully integrated team structure is not possible, it can be advantageous to have close, active collaborations.

4.3 Dead Experimental Codepaths

A common reaction to the hardening of glue code or pipeline jungles is that it becomes more and more tempting to perform experiments with alternative algorithms or tweaks by implementing these experimental codepaths as conditional

branches within the main production code. For any individual change, the cost of experimenting in this manner is relatively low—none of the surrounding infrastructure needs to be reworked. However, over time, these accumulated codepaths can create a growing debt. Maintaining backward compatibility with experimental codepaths is a burden for making more substantive changes. Furthermore, obsolete experimental codepaths can interact with each other in unpredictable ways, and tracking which combinations are incompatible quickly results in an exponential blowup in system complexity. A famous example of the dangers here was Knight Capital's system losing $465 million in 45 minutes apparently because of unexpected behavior from obsolete experimental codepaths [9].

As with the case of dead flags in traditional software [7], it is often beneficial to periodically reexamine each experimental branch to see what can be ripped out. Very often only a small subset of the possible branches is actually used; many others may have been tested once and abandoned.

Dead experimental codepaths are a symptom of a more fundamental issue: in a healthy machine learning system, experimental code should be well isolated, not leaving tendrils in multiple modules. This may require rethinking code APIs. In our experience, the kinds of things we want to experiment with vary over time; a redesign and a rewrite of some pieces may be needed periodically in order to move forward efficiently.

As a real-world anecdote, in a recent cleanup effort of one important machine learning system at Google, it was found possible to rip out tens of thousands of lines of unused experimental code-paths. A follow-on rewrite with a tighter API allowed experimentation with new algorithms to be performed with dramatically reduced effort and production risk and minimal incremental system complexity.

4.4 Configuration Debt

Another potentially surprising area where debt can accumulate is in the configuration of machine learning systems. Any large system has a wide range of configurable options, including which features are used, how data is selected, a wide variety of algorithm-specific learning settings, potential pre- or post-processing, verification methods, etc.

Many engineers do a great job of thinking hard about abstractions and unit tests in production code, but may treat configuration (and extension of configuration) as an afterthought. Indeed, verification or testing of configurations may not even be seen as important. Configuration by its very nature tends to be the place where re-

al-world messiness intrudes on beautiful algorithms.

Consider the following examples. Feature A was incorrectly logged from 9/14 to 9/17. Feature B is not available on data before 10/7. The code used to compute feature C has to change for data before and after 11/1 because of changes to the logging format. Feature D is not available in production, so a substitute features D` and D`` must be used when querying the model in a live setting. If feature Z is used, then jobs for training must be given extra memory due to lookup tables or they will train inefficiently. Feature Q precludes the use of feature R because of latency constraints. All this messiness makes configuration hard to modify correctly, and hard to reason about. However, mistakes in configuration can be costly, leading to serious loss of time, waste of computing resources, or production issues.

Also, in a mature system which is being actively developed, the number of lines of configuration can far exceed the number of lines of the code that actually does machine learning. Each line has a potential for mistakes, and configurations are by their nature ephemeral and less well tested.

Assertions about configuration invariants can be critical to prevent mistakes, but careful thought is needed about what kind of assertions will be useful. Another useful tool is the ability to present visual side-by-side differences (diffs) of two configurations. Because configurations are often copy-and-pasted with small modifications, such diffs highlight important changes. And clearly, configurations should be treated with the same level of seriousness as code changes, and be carefully code reviewed by peers.

5 Dealing with Changes in the External World

One of the things that makes machine learning systems so fascinating is that they often interact directly with the external world. Experience has shown that the external world is rarely stable. Indeed, the changing nature of the world is one of the sources of technical debt in machine learning systems.

5.1 Fixed Thresholds in Dynamic Systems

It is often necessary to pick a decision threshold for a given model to perform some action: to predict true or false, to mark an email as spam or not spam, to show or not show a given ad. One classic approach in machine learning is to choose a threshold from a set of possible thresholds, in order to get good tradeoffs on certain metrics, such as precision and recall. However, such thresholds are often manually set. Thus if a model updates on new data, the old manually set threshold may be invalid.

Manually updating many thresholds across many models is time-consuming and brittle.

A useful mitigation strategy for this kind of problem appears in [8], in which thresholds are learned via simple evaluation on heldout validation data.

5.2 When Correlations No Longer Correlate

Machine learning systems often have a difficult time distinguishing the impact of correlated features. This may not seem like a major problem: if two features are always correlated, but only one is truly causal, it may still seem okay to ascribe credit to both and rely on their observed co-occurrence.

However, if the world suddenly stops making these features co-occur, prediction behavior may change significantly. The full range of ML strategies for teasing apart correlation effects is beyond our scope; some excellent suggestions and references are given in [2]. For the purpose of this paper, we note that non-causal correlations are another source of hidden debt.

5.3 Monitoring and Testing

Unit testing of individual components and end-to-end tests of running systems are valuable, but in the face of a changing world such tests are not sufficient to provide evidence that a system is working as intended. Live monitoring of system behavior in real time is critical.

The key question is: what to monitor? It can be difficult to establish useful invariants, given that the purpose of machine learning systems is to adapt over time. We offer two reasonable starting points.

Prediction Bias. In a system that is working as intended, it should usually be the case that the distribution of predicted labels is equal to the distribution of observed labels. This is by no means a comprehensive test, as it can be met by a null model that simply predicts average values of label occurrences without regard to the input features. However, it is a surprisingly useful diagnostic, and changes in metrics such as this are often indicative of an issue that requires attention. For example, this method can help to detect cases in which the world behavior suddenly changes, making training distributions drawn from historical data no longer reflective of current reality. Slicing prediction bias by various dimensions isolate issues quickly, and can also be used for automated alerting.

Action Limits. In systems that are used to take actions in the real world, it can be useful to set and enforce action limits as a sanity check. These limits should be broad enough not to trigger spuriously. If the system hits a limit for a given action, automated alerts should fire and trigger manual intervention or investigation.

6 Conclusions: Paying it Off

This paper has highlighted a number of areas where machine learning systems can create technical debt, sometimes in surprising ways. This is not to say that machine learning is bad, or even that technical debt is something to be avoided at all costs. It may be reasonable to take on moderate technical debt for the benefit of moving quickly in the short term, but this must be recognized and accounted for lest it quickly grow unmanageable.

Perhaps the most important insight to be gained is that technical debt is an issue that both engineers and researchers need to be aware of. Research solutions that provide a tiny accuracy benefit at the cost of massive increases in system complexity are rarely wise practice. Even the addition of one or two seemingly innocuous data dependencies can slow further progress.

Paying down technical debt is not always as exciting as proving a new theorem, but it is a critical part of consistently strong innovation. And developing holistic, elegant solutions for complex machine learning systems is deeply rewarding work.

Acknowledgments

This paper owes much to the important lessons learned day to day in a culture that values both innovative ML research and strong engineering practice. Many colleagues have helped shape our thoughts here, and the benefit of accumulated folk wisdom cannot be overstated. We would like to specifically recognize the following: Luis Cobo, Sharat Chikkerur, Jean-Francois Crespo, Jeff Dean, Dan Dennison, Philip Henderson, Arnar Mar Hrafnkelsson, Ankur Jain, Joe Kovac, Jeremy Kubica, H. Brendan McMahan, Satyaki Mahalanabis, Lan Nie, Michael Pohl, Abdul Salem, Sajid Siddiqi, Ricky Shan, Alan Skelly, Cory Williams, and Andrew Young.

Why It's Often Easier To Innovate In China Than In The United States

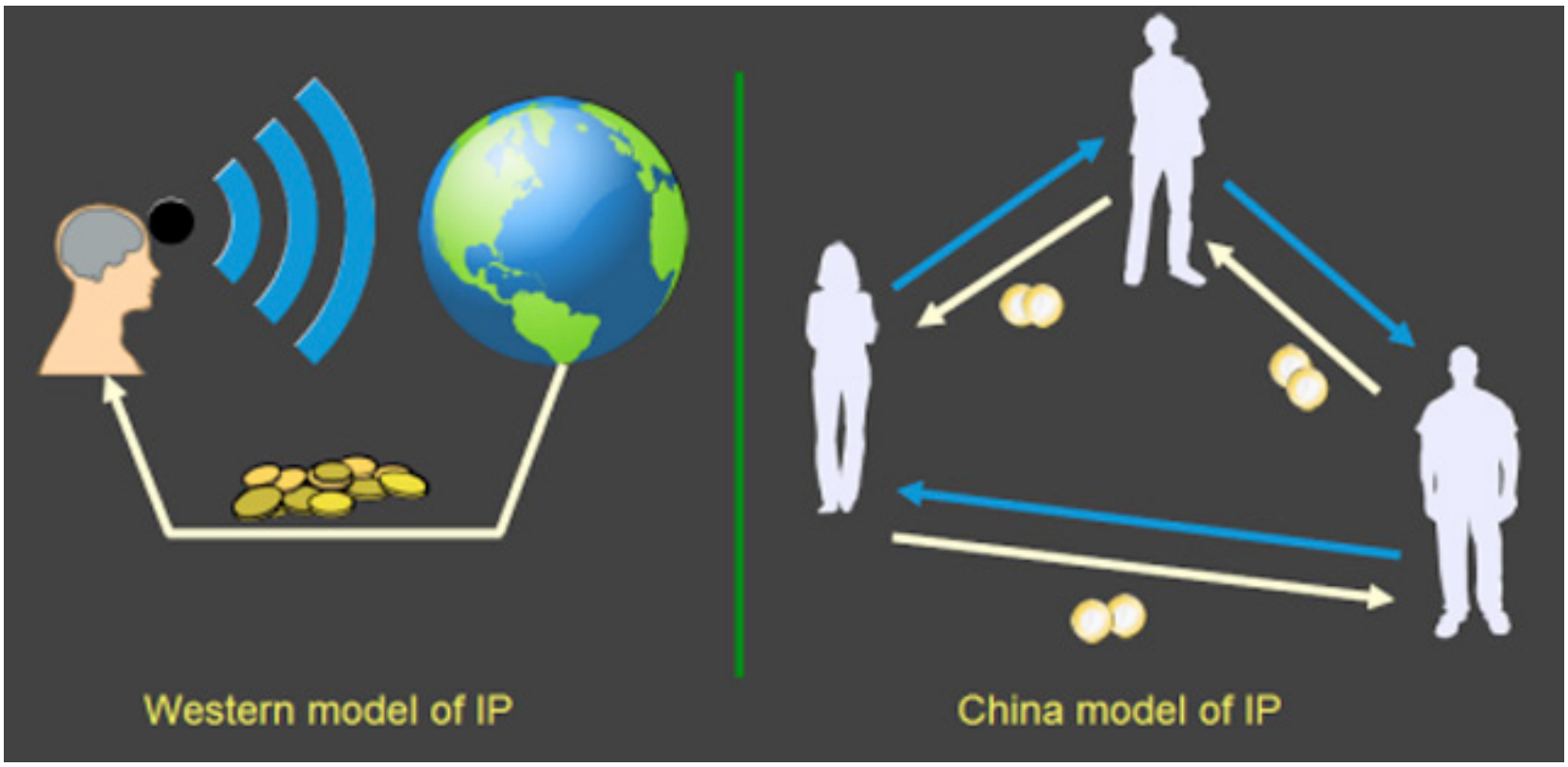
Andrew 'bunnie' Huang, Gizmodo, June 15 2015

About a year and a half ago, I discovered this $12 "Gongkai" cell phone (pictured above) in the markets of Shenzhen, China. My most striking impression was that Chinese entrepreneurs had relatively unfettered access to cutting-edge technology, enabling start-ups to innovate while bootstrapping. Meanwhile, Western entrepreneurs often find themselves trapped in a spiderweb of IP frameworks, spending more money on lawyers than on tooling.

Further investigation taught me that the Chinese have a parallel system of traditions and ethics around sharing IP, which lead me to coin the term "gongkai".

This is deliberately not the Chinese word for "Open Source", because that word (kaiyuan) refers to openness in a Western-style IP framework, which this not. Gongkai is more a reference to the fact that copyrighted documents, sometimes labeled "confidential" and "proprietary", are made known to the public and shared overtly, but not necessarily according to the letter of the law. However, this copying isn't a one-way flow of value, as it would be in the case of copied movies or music. Rather, these documents are the knowledge base needed to build a phone using the copy-

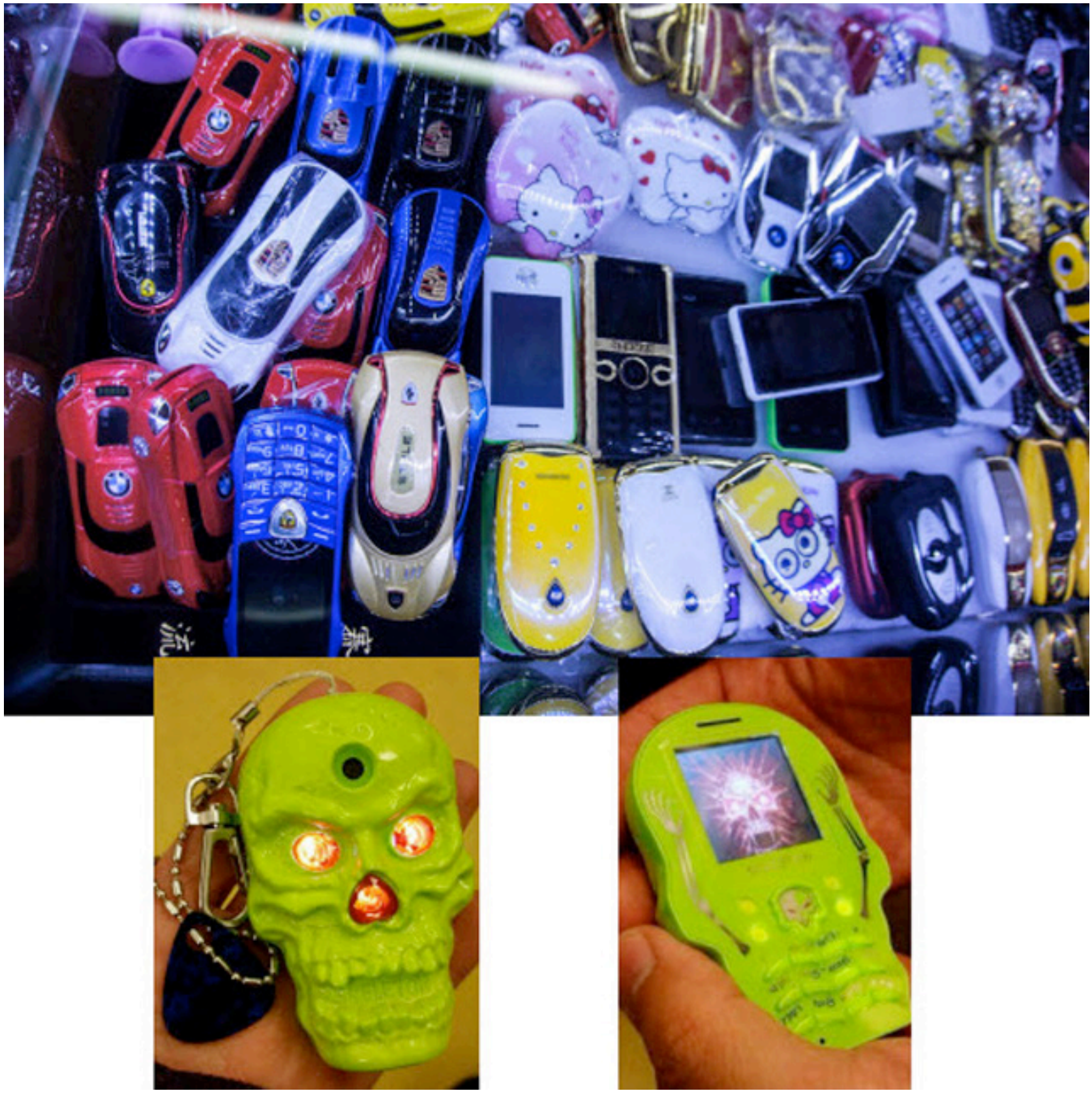
right owner's chips, and as such, this sharing of documents helps to promote the sales of their chips. There is ultimately, if you will, a quid-pro-quo between the copyright holders and the copiers.



This fuzzy, gray relationship between companies and entrepreneurs is just one manifestation of a much broader cultural gap between the East and the West. The West has a "broadcast" view of IP and ownership: good ideas and innovation are credited to a clearly specified set of authors or inventors, and society pays them a royalty for their initiative and good works. China has a "network" view of IP and ownership: the far-sight necessary to create good ideas and innovations is attained by standing on the shoulders of others, and as such there is a network of people who trade these ideas as favors among each other. In a system with such a loose attitude toward IP, sharing with the network is necessary as tomorrow it could be your friend standing on your shoulders, and you'll be looking to them for favors. This is unlike the West, where rule of law enables IP to be amassed over a long period of time, creating impenetrable monopoly positions. It's good for the guys on top, but tough for the upstarts.

This brings us to the situation we have today: Apple and Google are building amazing phones of outstanding quality, and start-ups can only hope to build an appcessory for their ecosystem. I've reviewed business plans of over a hundred hardware startups by now, and most of them are using overpriced chipsets built using antiquated process technologies as their foundation. I'm no exception to this rule – we use the Freescale i.MX6 for Novena, which is neither the cheapest nor the fastest chip on the market, but it is the one chip where anyone can freely download almost complete documentation and anyone can buy it on Digikey. This parallel constraint of scarce documentation and scarce supply for cutting edge technology forces Western hardware entrepreneurs to look primarily at Arduino, Beaglebone and

Raspberry Pi as starting points for their good ideas.



Above: Every object pictured is a phone. Inset: detail of the “Skeleton” novelty phone. Image credits: Halfdan, Rachel Kalmar

Chinese entrepreneurs, on the other hand, churn out new phones at an almost alarming pace. Phone models change on a seasonal basis. Entrepreneurs experiment all the time, integrating whacky features into phones, such as cigarette lighters, extra-large battery packs (that can be used to charge another phone), huge buttons (for the visually impaired), reduced buttons (to give to children as emergency-call phones), watch form factors, and so forth. This is enabled because very small teams of engineers can obtain complete design packages for working phones – case, board, and firmware – allowing them to fork the design and focus only on the pieces they really care about.

As a hardware engineer, I want that. I want to be able to fork existing cell phone designs. I want to be able to use a 364 MHz 32-bit microcontroller with megabytes of integrated RAM and dozens of peripherals costing $3 in single quantities, instead of a 16 MHz 8-bit microcontroller with a few kilobytes of RAM and a smattering of peripherals costing $6 in single quantities. Unfortunately, queries into getting a Western-licensed EDK for the chips used in the Chinese phones were met with a cold shoulder – our volumes are too small, or we have to enter minimum purchase agreements backed by hundreds of thousands of dollars in a cash deposit; and even then, these EDKs don't include all the reference material the Chinese get to play with. The datasheets are incomplete and as a result you're forced to use their proprietary OS ports. It feels like a case of the nice guys finishing last. Can we find a way to still get ahead, yet still play nice?

We did some research into the legal frameworks and challenges around absorbing Gongkai IP into the Western ecosystem, and we believe we've found a path to repatriate some of the IP from Gongkai into proper Open Source. However, I must interject with a standard disclaimer: we're not lawyers, so we'll tell you our beliefs but don't construe them as legal advice. Our intention is to exercise our right to reverse engineer in a careful, educated fashion to increase the likelihood that, if push comes to shove, the courts will agree with our actions. However, we also feel that shying away from reverse engineering simply because it's controversial is a slippery slope: you must exercise your rights to have them. If women didn't vote and black people sat in the back of the bus because they were afraid of controversy, the US would still be segregated and without universal suffrage.

Sometimes, you just have to stand up and assert your rights.

There are two broad categories of issues we have to deal with, patents and copyrights. For patents, the issues are complex, yet it seems the most practical approach is to essentially punt on the issue. This is what the majority of the open source community does, and in fact many corporations have similar policies at the engineering level. Nobody, as far as we know, checks their Linux commits for patent infringement before upstreaming them. Why? Among other reasons, it takes a huge amount of resources to determine which patents apply, and if one could be infringing; and even after expending those resources, one cannot be 100% sure. Furthermore, if one becomes very familiar with the body of patents, it amplifies the possibility that an infringement, should it be found, is willful and thus triple damages. Finally, it's not even clear where the liability lies, particularly in an open source context. Thus, we do our best not to infringe, but cannot be 100% sure that no one will allege infringement. However, we do apply a license to our work which has a "poison pill" clause

for patent holders that do attempt to litigate.

For copyrights, the issue is also extremely complex. The EFF's Coders' Rights Project has a Reverse Engineering FAQ that's a good read if you really want to dig into the issues. The tl;dr is that courts have found that reverse engineering to understand the ideas embedded in code and to achieve interoperability is fair use. As a result, we have the right to study the Gongkai-style IP, understand it, and produce a new work to which we can apply a Western-style Open IP license. Also, none of the files or binaries were encrypted or had access controlled by any technological measure – no circumvention, no DMCA problem.

Furthermore, all the files were obtained from searches linking to public servers – so no CFAA problem, and none of the devices we used in the work came with shrink-wraps, click-throughs, or other end-user license agreements, terms of use, or other agreements that could waive our rights.

Thus empowered by our fair use rights, we decided to embark on a journey to reverse engineer the Mediatek MT6260. It's a 364 MHz, ARM7EJ-S, backed by 8MiB of RAM and dozens of peripherals, from the routine I2C, SPI, PWM and UART to tantalizing extras like an LCD + touchscreen controller, audio codec with speaker amplifier, battery charger, USB, Bluetooth, and of course, GSM. The gray market prices it around $3/unit in single quantities. You do have to read or speak Chinese to get it, and supply has been a bit spotty lately due to high Q4 demand, but we're hoping the market will open up a bit as things slow down for Chinese New Year.

For a chip of such complexity, we don't expect our two-man team to be able to unravel its entirety working on it as a part-time hobby project over the period of a year. Rather, we'd be happy if we got enough functionality so that the next time we reach for an ATMega or STM32, we'd also seriously consider the MT6260 as an alternative. Thus, we set out as our goal to port NuttX, a BSD-licensed RTOS, to the chip, and to create a solid framework for incrementally porting drivers for the various peripherals into NuttX. Accompanying this code base would be original hardware schematics, libraries and board layouts that are licensed using CC BY-SA-3.0 plus an Apache 2.0 rider for patent issues.

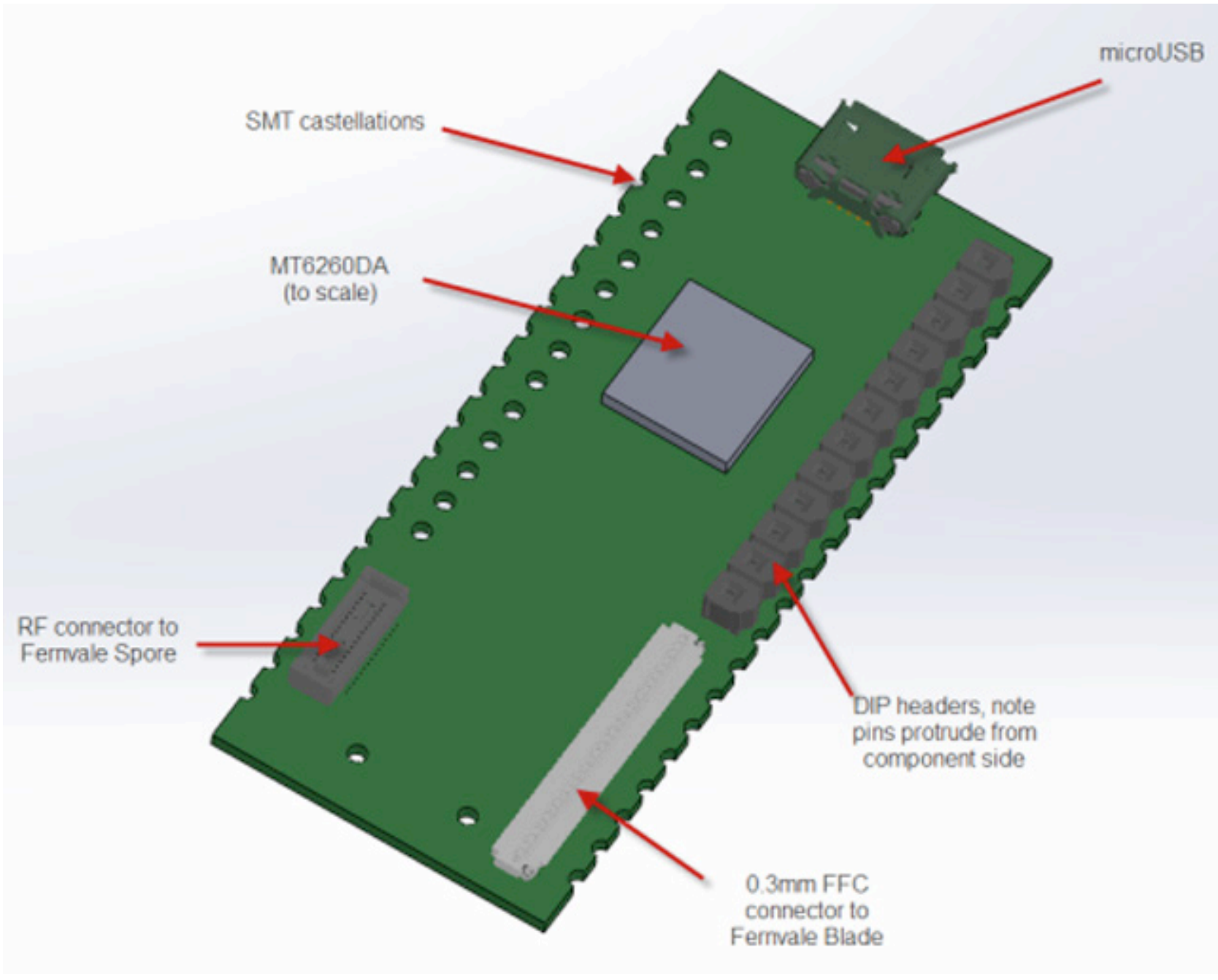
And thus, the Fernvale project was born.

Fernvale Hardware

Compared to the firmware, the hardware reverse engineering task was fairly straightforward. The documents we could scavenge gave us a notion of the ball-out

for the chip, and the naming scheme for the pins was sufficiently descriptive that I could apply common sense and experience to guess the correct method for connecting the chip. For areas that were ambiguous, we had some stripped down phones I could buzz out with a multimeter or stare at under a microscope to determine connectivity; and in the worst case I could also probe a live phone with an oscilloscope just to make sure my understanding was correct.

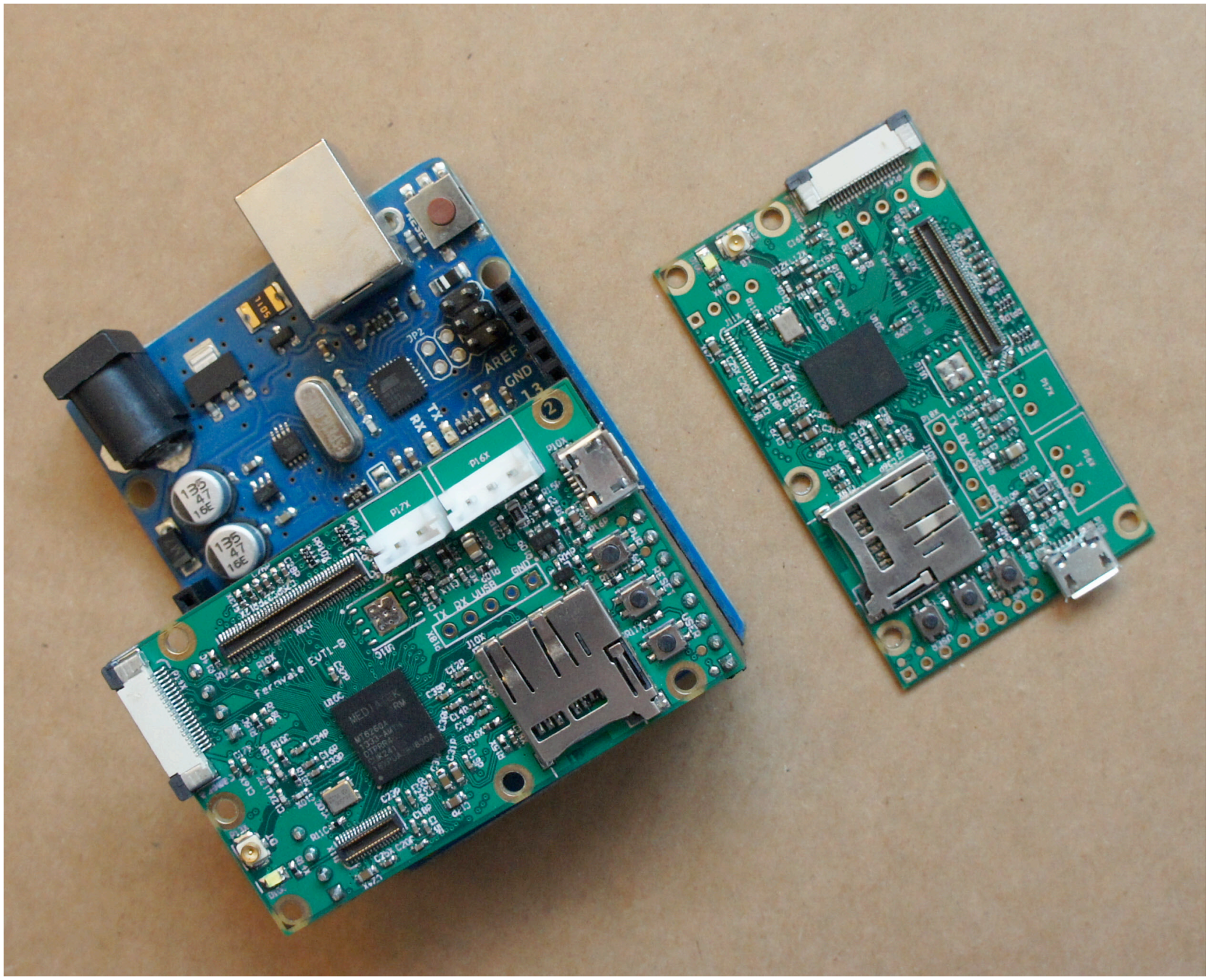
The more difficult question was how to architect the hardware. We weren't gunning to build a phone – rather, we wanted to build something a bit closer to the Spark Core, a generic SoM that can be used in various IoT-type applications. In fact, our original renderings and pin-outs were designed to be compatible with the Spark ecosystem of hardware extensions, until we realized there were just too many interesting peripherals in the MT6260 to fit into such a small footprint.



Above: early sketches of the Fernvale hardware

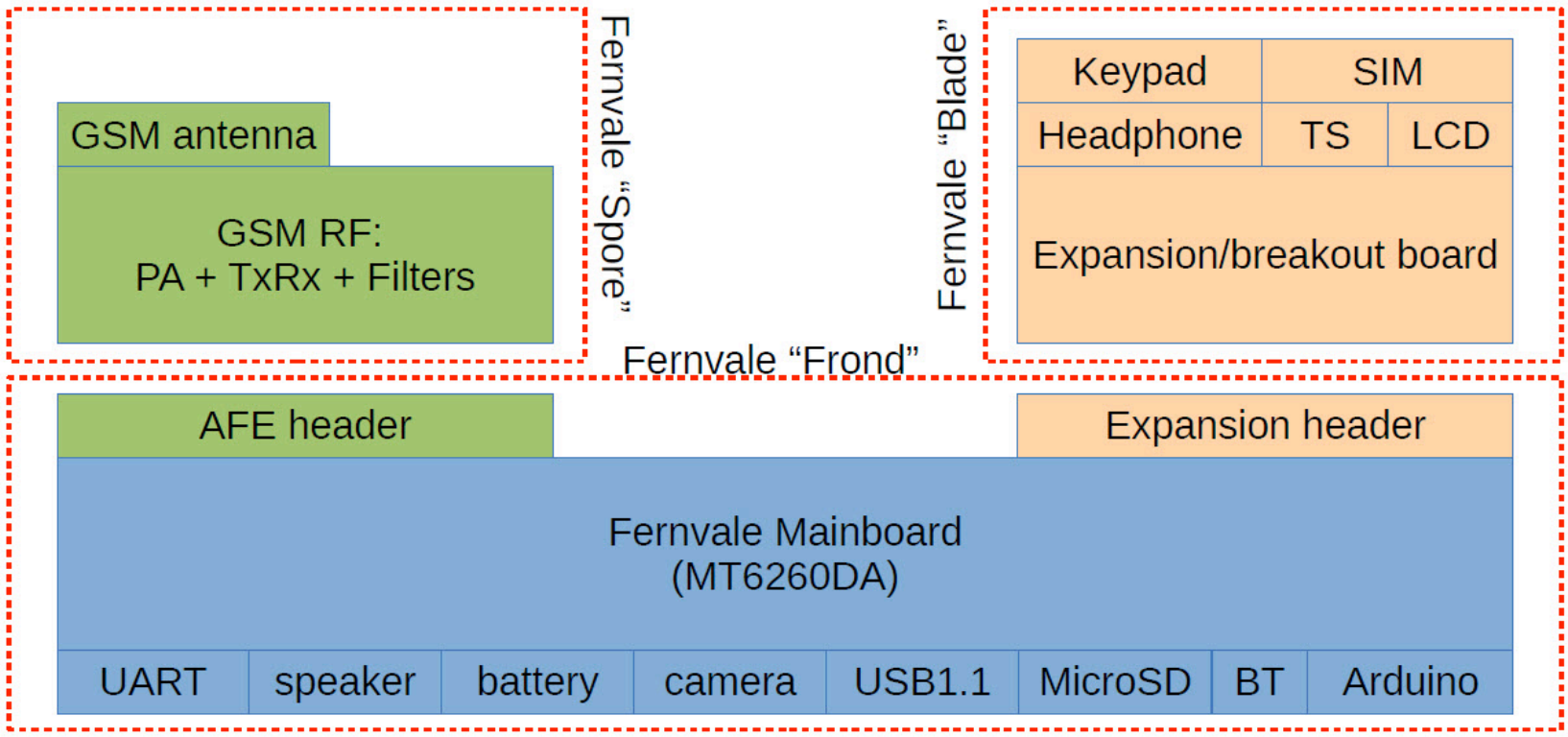
We settled eventually upon a single-sided core PCB that we call the "Fernvale Frond" which embeds the microUSB, microSD, battery, camera, speaker, and Bluetooth functionality (as well as the obligatory buttons and LED). It's slim, at 3.5mm thick,

and at 57x35mm it's also on the small side. We included holes to mount a partial set of pin headers, spaced to be compatible with an Arduino, although it can only be plugged into 3.3V-compatible Arduino devices.

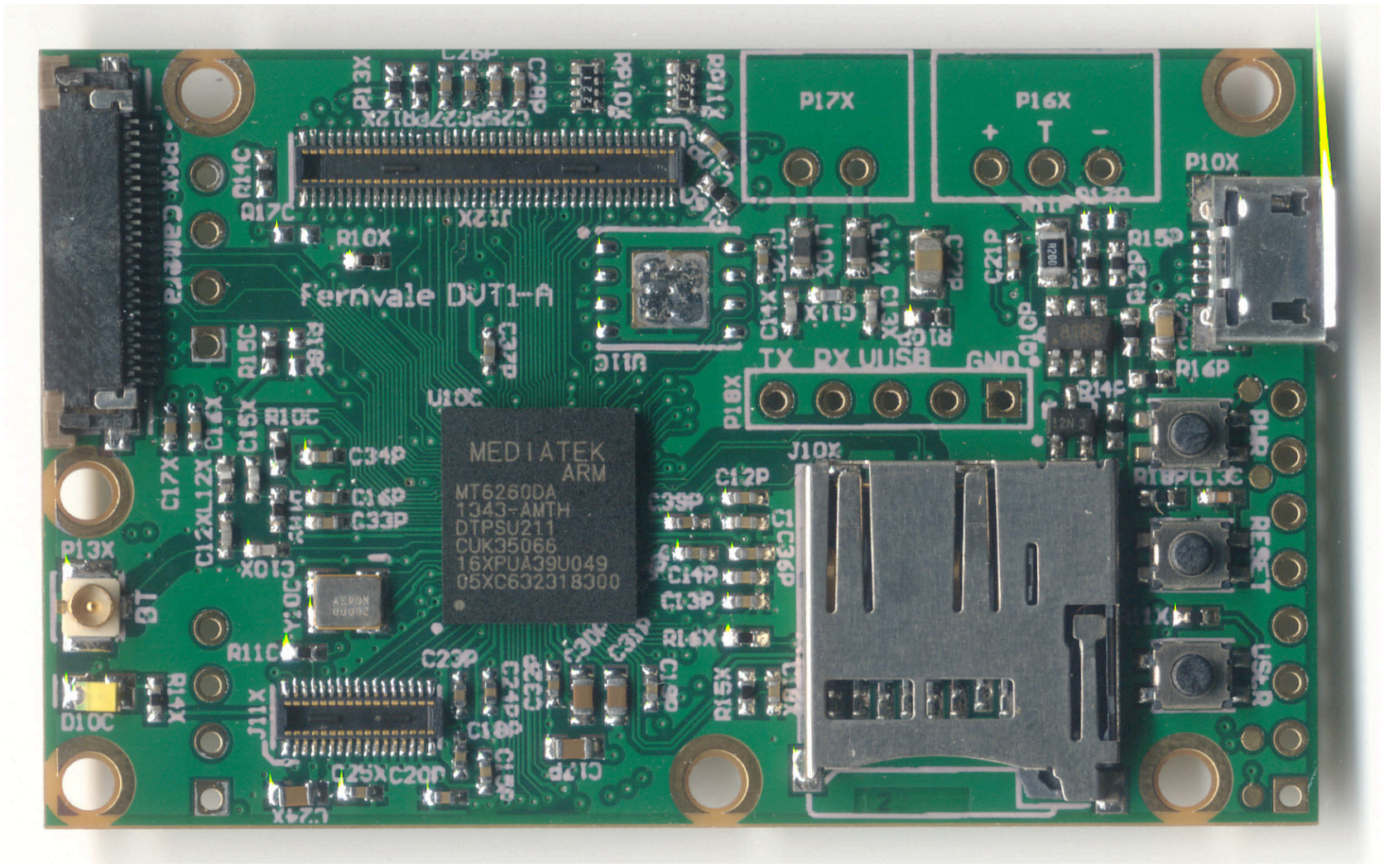


Above: actual implementation of Fernvale, pictured with Arduino for size reference

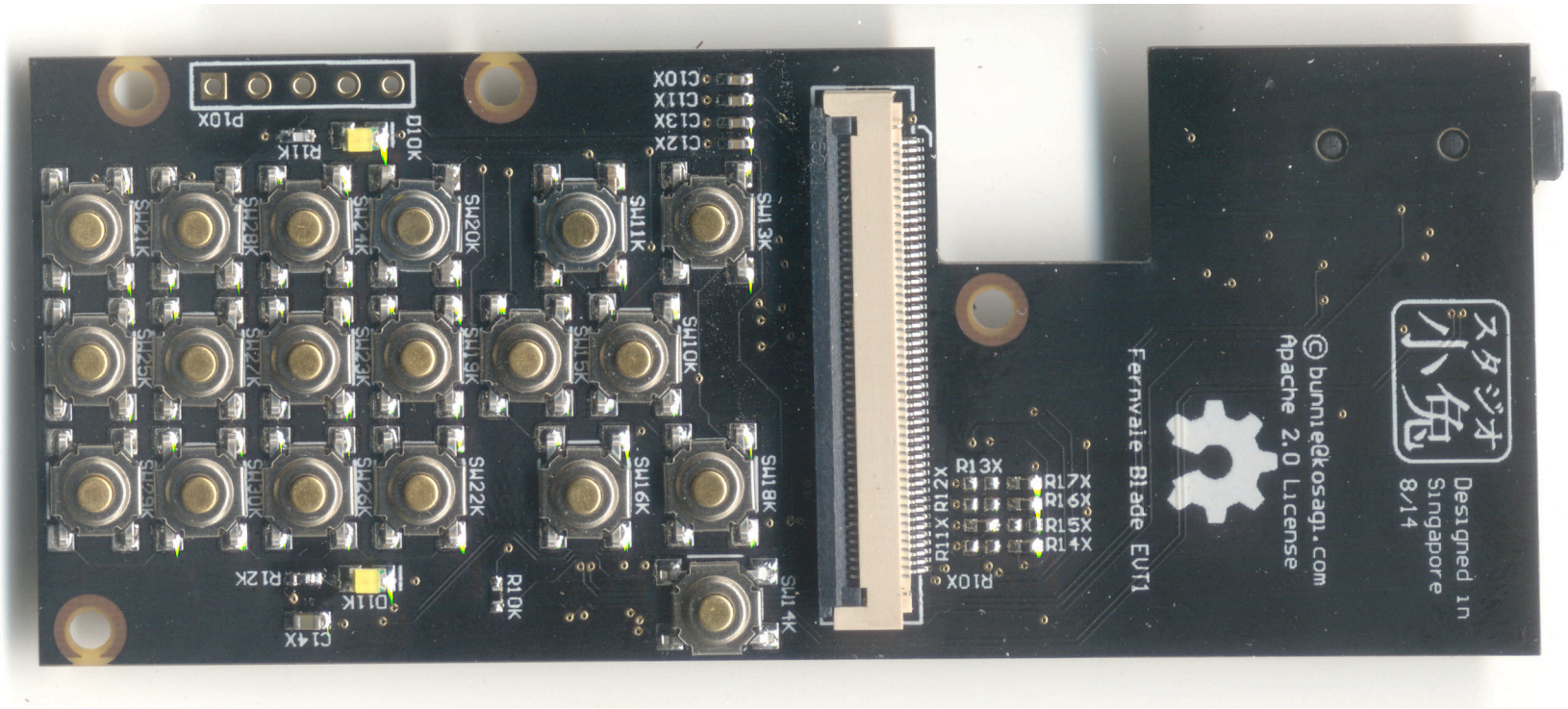
The remaining peripherals are broken out to a pair of connectors. One connector is dedicated to GSM-related signals; the other to UI-related peripherals. Splitting GSM into a module with many choices for the RF front end is important, because it makes GSM a bona-fide user-installed feature, thus pushing the regulatory and emissions issue down to the user level. Also, splitting the UI-related features out to another board costs down the core module, so it can fit into numerous scenarios without locking users into a particular LCD or button arrangement.



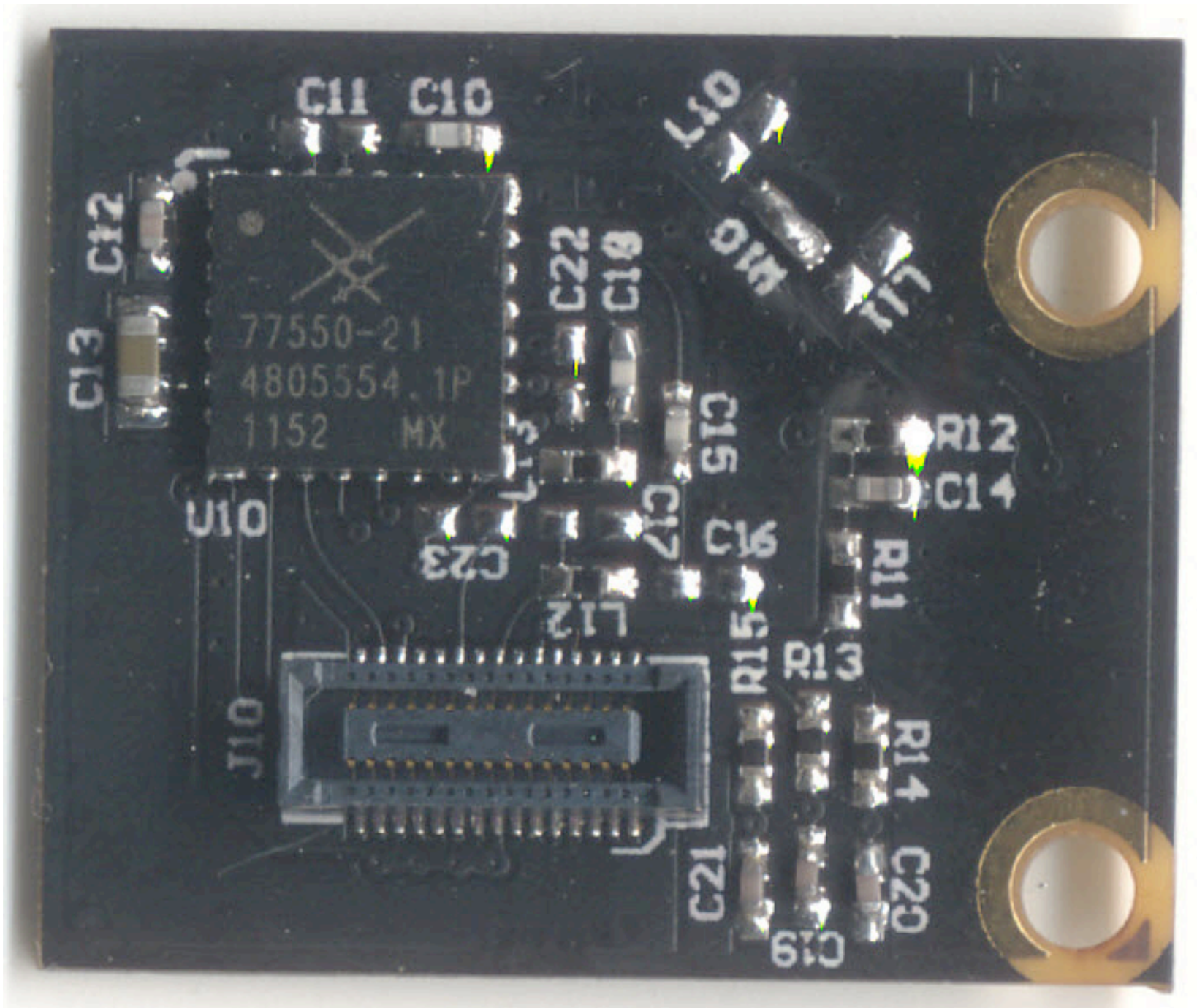
Above: Fernvale system diagram, showing the features of each of the three boards



Fernvale Frond mainboard



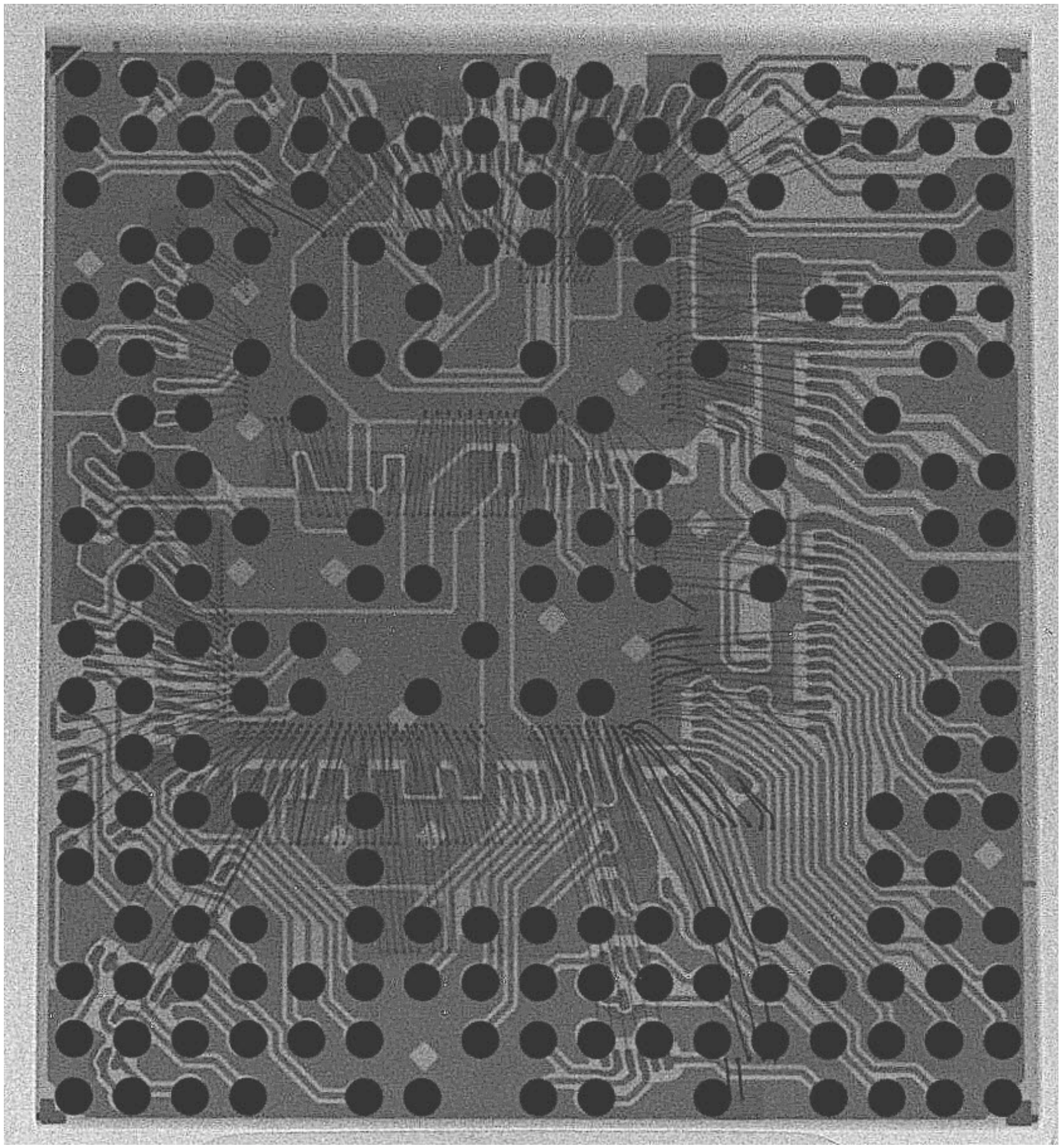
Fernvale blade UI breakout



Fernvale spore AFE dev board

All the hardware source documents can be downloaded from our wiki.

As an interesting side-note, I had some X-rays taken of the MT6260. We did this to help us identify fake components, just in case we encountered units being sold as empty epoxy blocks, or as remarked versions of other chips (the MT6260 has variants, such as the -DA and the -A, the difference being how much on-chip FLASH is included).



X-ray of the MT6260 chip. A sharp eye can pick out the outline of multiple ICs among the wirebonds. Image credit: Nadya Peek

To our surprise, this $3 chip didn't contain a single IC, but rather, it's a set of at least

4 chips, possibly 5, integrated into a single multi-chip module (MCM) containing hundreds of wire bonds. I remember back when the Pentium Pro's dual-die package came out. That sparked arguments over yielded costs of MCMs versus using a single bigger die; generally, multi-chip modules were considered exotic and expensive. I also remember at the time, Krste Asanović, then a professor at the MIT AI Lab now at Berkeley, told me that the future wouldn't be system on a chip, but rather "system mostly on a chip". The root of his claim is that the economics of adding in mask layers to merge DRAM, FLASH, Analog, RF, and Digital into a single process wasn't favorable, and instead it would be cheaper and easier to bond multiple die together into a single package. It's a race between the yield and cost impact (both per-unit and NRE) of adding more process steps in the semiconductor fab, vs. the yield impact (and relative reworkability and lower NRE cost) of assembling modules. Single-chip SoCs was the zeitgeist at the time (and still kind of is), so it's interesting to see a significant datapoint validating Krste's insight.

Reversing the Boot Structure

The amount of documentation made available to Shanzhai engineers in China seems to be just enough to enable them to assemble a phone and customize its UI, but not enough to do a full OS port. You eventually come to recognize that all the phones based on a particular chipset have the same backdoor codes, and often times the UI is inconsistent with the implemented hardware. For example, the $12 phone mentioned at the top of the post will prompt you to plug headphones into the headphone jack for the FM radio to work, yet there is no headphone jack provided in the hardware. In order to make Fernvale accessible to engineers in the West, we had to reconstruct everything from scratch, from the toolchain, to the firmware flashing tool, to the OS, to the applications. Given that all the Chinese phone implementations simply rely upon Mediatek's proprietary toolchain, we had to do some reverse engineering work to figure out the boot process and firmware upload protocol.

My first step is always to dump the ROM, if possible. We found exactly one phone model which featured an external ROM that we could desolder (it uses the -D ROM-less variant of the chip), and we read its contents using a conventional ROM reader. The good news is that we saw very little ciphertext in the ROM; the bad news is there's a lot of compressed data. Below is a page from our notes after doing a static analysis on the ROM image.

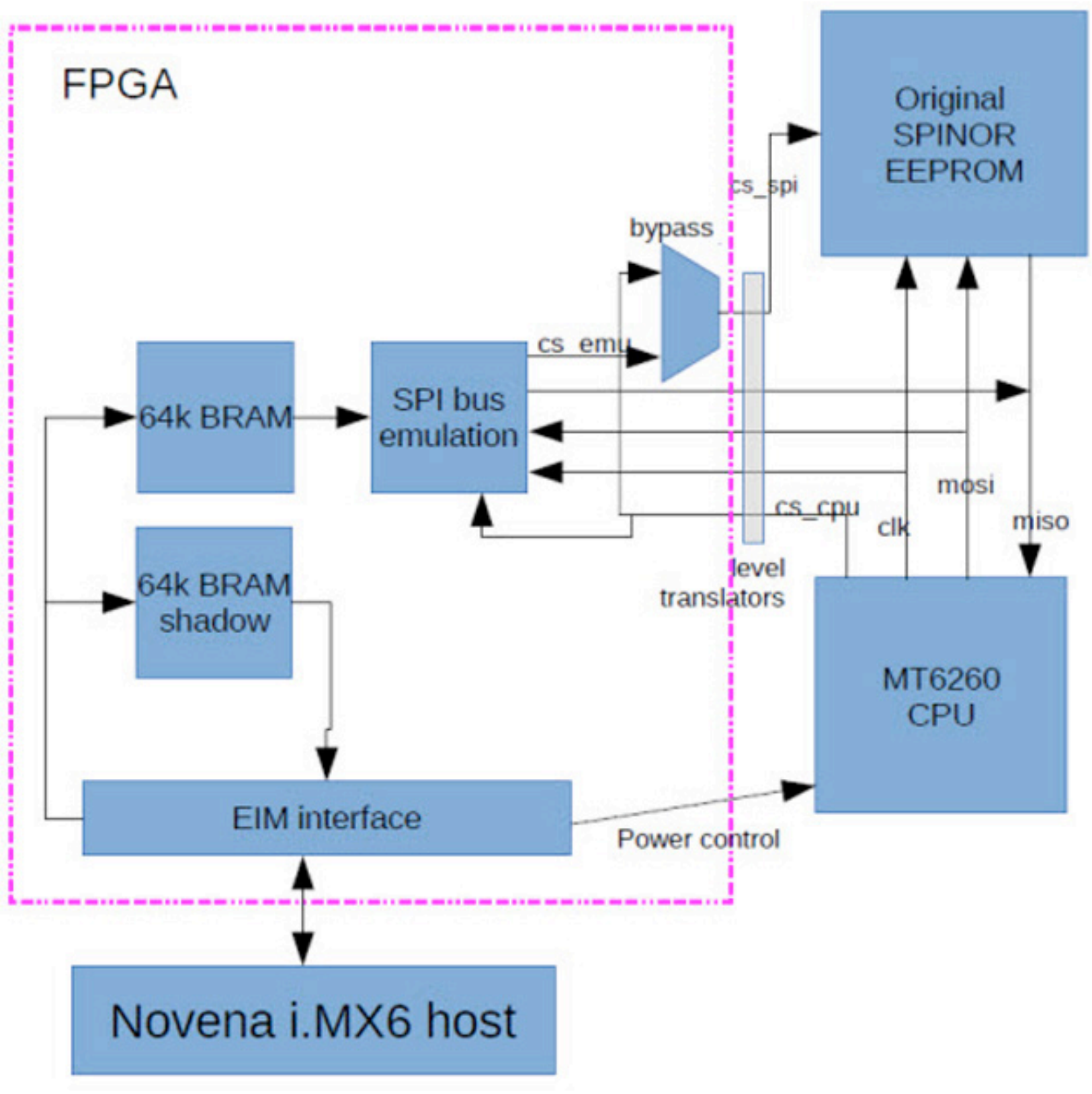
```
0x0000_0000 media signature “SF_BOOT” 0x0000_0200 bootloader signature
“BRLYT”, “BBBB” 0x0000_0800 sector header 1 (“MMM.8”) 0x0000_09BC reset
vector table 0x0000_0A10 start of ARM32 instructions – stage 1 bootload-
er? 0x0000_3400 sector header 2 (“MMM.8”) – stage 2 bootloader? 0x0000_
```

```
A518 thunk table of some type 0x0000_B704 end of code (padding until
next sector) 0x0001_0000 sector header 3( “MMM.8”) – kernel? 0x0001_0368
jump table + runtime setup (stack, etc.) 0x0001_0828 ARM thumb code
start – possibly also baseband code 0x0007_2F04 code end 0x0007_2F05 –
0x0009_F0005 padding “DFFF” 0x0009_F006 code section begin “Accelerated
Technology / ATI / Nucleus PLUS” 0x000A_2C1A code section end; pad with
zeros 0x000A_328C region of compressed/unknown data begin 0x007E_E200
modified FAT partition #1 0x007E_F400 modified FAT partition #2
```

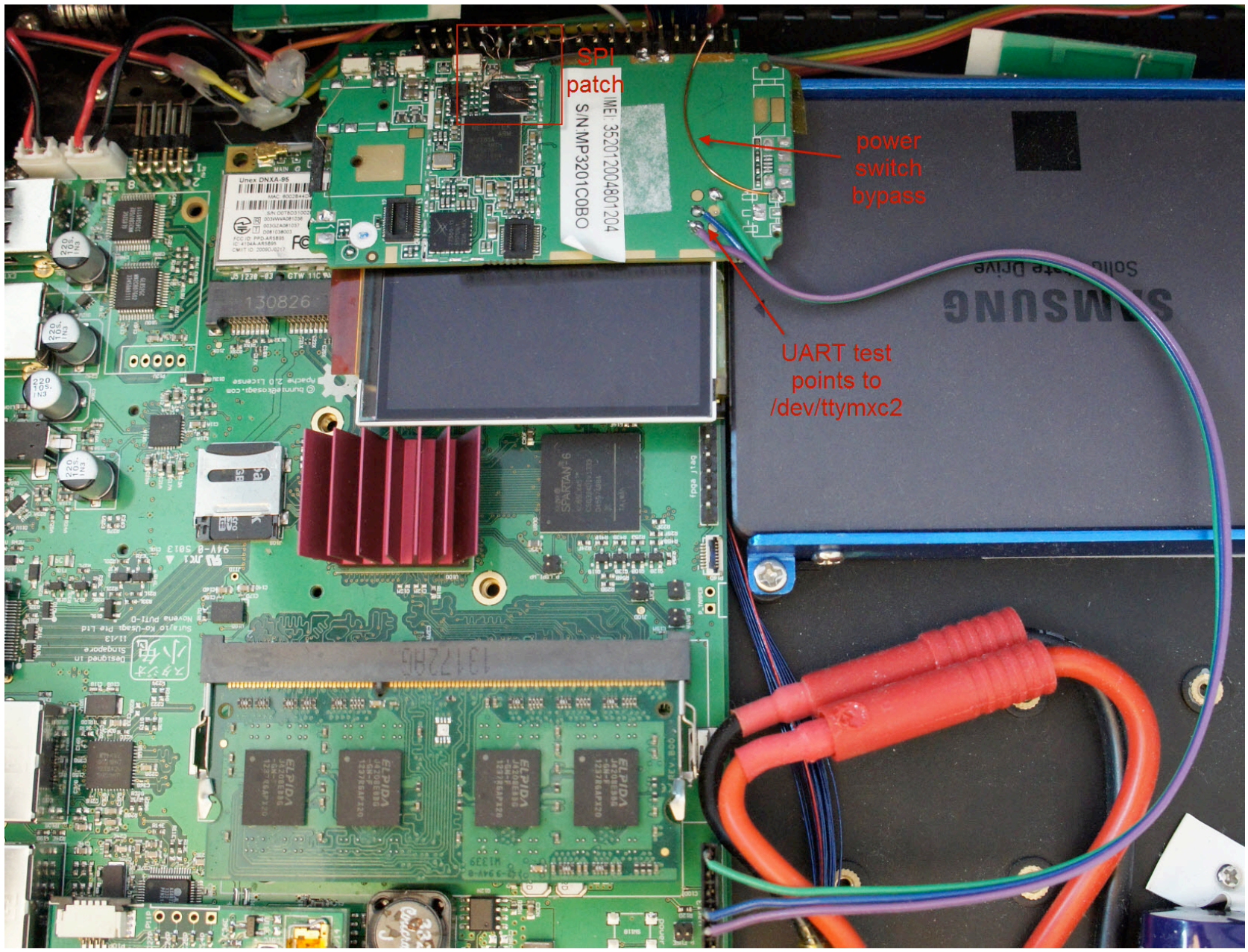
One concern about reverse engineering SoCs is that they have an internal boot ROM that is always run before code is loaded from an external device. This internal ROM can also have signature and security checks that prevent tampering with the external code, and so to determine the effort level required we wanted to quickly figure out how much code was running inside the CPU before jumping to external boot code. This task was made super-quick, done in a couple hours, using a Tek MDO4104B-6. It has the uncanny ability to take deep, high-resolution analog traces and do post-capture analysis as digital data. For example, we could simply probe around while cycling power until we saw something that looked like RS-232, and then run a post-capture analysis to extract any ASCII text that could be coded in the analog traces. Likewise, we could capture SPI traces and the oscilloscope could extract ROM access patterns through a similar method. By looking at the timing of text emissions versus SPI ROM address patterns, we were able to quickly determine that if the internal boot ROM did any verification, it was minimal and nothing approaching the computational complexity of RSA.

Above: Screenshot from the Tek MDO4104B-6, showing the analog trace in yellow, and the ASCII data extracted in cyan. The top quarter shows a zoomed-out view of the entire capture; one can clearly see how SPI ROM accesses in gray are punctuated with console output in cyan.

From here, we needed to speed up our measure-modify-test loop; desoldering the ROM, sticking it in a burner, and resoldering it onto the board was going to get old really fast. Given that we had previously implemented a NAND FLASH ROMulator on Novena, it made sense to re-use that code base and implement a SPI ROMulator. We hacked up a GPBB board and its corresponding FPGA code, and implemented the ability to swap between the original boot SPI ROM and a dual-ported 64kiB emulator region that is also memory-mapped into the Novena Linux host's address space.



Block diagram of the SPI ROMulator FPGA



There's a phone in my Novena! What's that doing there?

A combination of these tools – the address stream determined by the Tek oscilloscope, rapid ROM patching by the ROMulator, and static code analysis using IDA (we found a SHA-1 implementation) – enabled us to determine that the initial bootloader, which we refer to as the 1bl, was hash-checked using a SHA-1 appendix.

Building a Beachhead

The next step was to create a small interactive shell which we could use as a beachhead for running experiments on the target hardware. Xobs created a compact REPL environment called Fernly which supports commands like peeking and poking to memory, and dumping CPU registers.

Because we designed the ROMulator to make the emulated ROM appear as a 64k memory-mapped window on a Linux host, it enables the use a variety of POSIX abstractions, such as mmap(), open() (via /dev/mem), read() and write(), to access the emulated ROM. xobs used these abstractions to create an I/O target for radare2. The

I/O target automatically updates the SHA-1 hash every time we made changes in the 1bl code space, enabling us to do cute things like interactively patch and disassemble code within the emulated ROM space.

We also wired up the power switch of the phone to an FPGA I/O, so we could write automated scripts that toggle the power on the phone while updating the ROM contents, allowing us to do automated fuzzing of unknown hardware blocks.

Attaching a Debugger

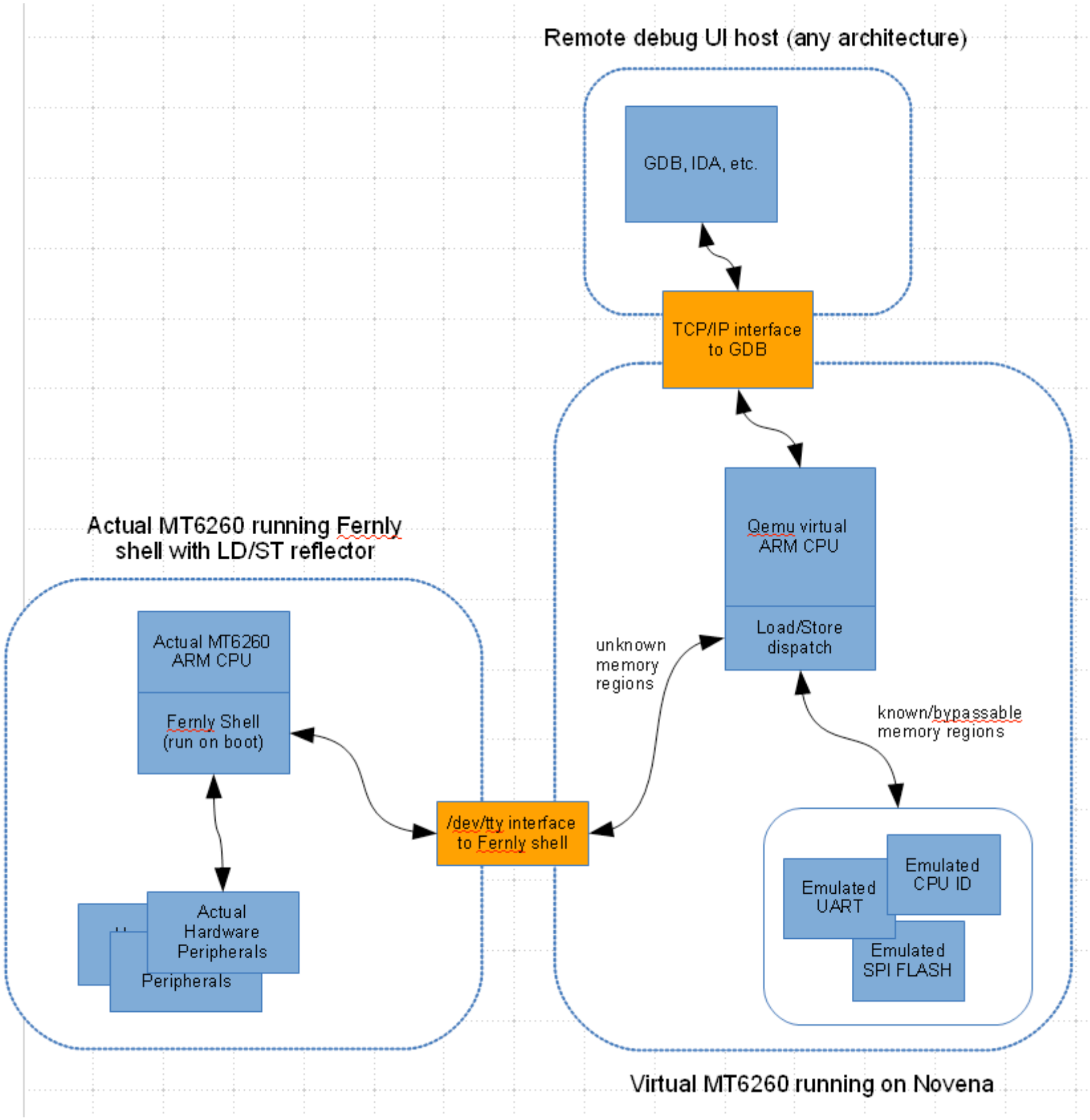
Because of the difficulty in trying to locate critical blocks, and because JTAG is mul-

tiplexed with critical functions on the target device, an unconventional approach was taken to attach a debugger: xobs emulates the ARM core, and uses his fernly shell to reflect virtual loads and stores to the live target. This allows us to attach a remote debugger to the emulated core, bypassing the need for JTAG and allowing us to use cross-platform tools such as IDA on x86 for the reversing UI.

At the heart of this technique is Qemu, a multi-platform system emulator. It supports emulating ARM targets, specifically the ARMv5 used in the target device. A new machine type was created called “fernvale” that implements part of the observed hardware on the target, and simply passes unknown memory accesses directly to the device.

The Fernly shell was stripped down to only support three commands: write, read, and zero-memory. The write command pokes a byte, word, or dword into RAM on the live target, and a read command reads a byte, word, or dword from the live target. The zero-memory command is an optimization, as the operating system writes large quantities of zeroes across a large memory area.

In addition, the serial port registers are hooked and emulated, allowing a host system to display serial data as if it were printed on the target device. Finally, SPI, IRAM, and PSRAM are all emulated as they would appear on the real device. Other areas of memory are either trapped and funneled to the actual device, or are left unmapped and are reported as errors by Qemu.



The diagram above illustrates the architecture of the debugger.

Invoking the debugger is a multi-stage process. First, the actual MT6260 target is primed with the Fernly shell environment. Then, the Qemu virtual ARM CPU is "booted" using the original vendor image – or rather, primed with a known register state at a convenient point in the boot process. At this point, code execution proceeds on the virtual machine until a load or store is performed to an unknown address. Virtual machine execution is paused while a query is sent to the real MT6260 via the Fernly shell interface, and the load or store is executed on the real machine. The results of this load or store is then relayed to the virtual machine and execution is resumed. Of course, Fernly will crash if a store happens to land somewhere inside its memory footprint. Thus, we had to hide the Fernly shell code in a region of IRAM that's trapped and emulated, so loads and stores don't overwrite the shell code. Run-

ning Fernly directly out of the SPI ROM also doesn't work as part of the initialization routine of the vendor binary modifies SPI ROM timings, causing SPI emulation to fail.

Emulating the target CPU allows us to attach a remote debugger (such as IDA) via GDB over TCP without needing to bother with JTAG. The debugger has complete control over the emulated CPU, and can access its emulated RAM. Furthermore, due to the architecture of qemu, if the debugger attempts to access any memory-mapped IO that is redirected to the real target, the debugger will be able to display live values in memory. In this way, the real target hardware is mostly idle, and is left running in the Fernly shell, while the virtual CPU performs all the work. The tight integration of this package with IDA-over-GDB also allows us to very quickly and dynamically execute subroutines and functions to confirm their purpose.

Below is an example of the output of the hybrid Qemu/live-target debug harness. You can see the trapped serial writes appearing on the console, plus a log of the writes and reads executed by the emulated ARM CPU, as they are relayed to the live target running the reduced Fernly shell.

```
bunnie@bunnie-novena-laptop:~/code/fernvale-qemu$ ./run.sh ~~~ Welcome
to MTK Bootloader V005 (since 2005) ~~~ **========================
======================** READ WORD Fernvale Live 0xa0010328 = 0x0000...
ok WRITE WORD Fernvale Live 0xa0010328 = 0x0800... ok READ WORD Fern-
vale Live 0xa0010230 = 0x0001... ok WRITE WORD Fernvale Live 0xa0010230
= 0x0001... ok READ DWORD Fernvale Live 0xa0020c80 = 0x11111011... ok
WRITE DWORD Fernvale Live 0xa0020c80 = 0x11111011... ok READ DWORD
Fernvale Live 0xa0020c90 = 0x11111111... ok WRITE DWORD Fernvale Live
0xa0020c90 = 0x11111111... ok READ WORD Fernvale Live 0xa0020b10 =
0x3f34... ok WRITE WORD Fernvale Live 0xa0020b10 = 0x3f34... ok
```

From this beachhead, we were able to discover the offsets of a few IP blocks that were re-used from previous known Mediatek chips (such as the MT6235 in the osmocomBB http://bb.osmocom.org/trac/wiki/MT62... by searching for their "signature". The signature ranged from things as simple as the power-on default register values, to changes in bit patterns due to the side effects of bit set/clear registers located at offsets within the IP block's address space. Using this technique, we were able to find the register offsets of several peripherals.

0x00000000	0x0fffffff	0x0fffffff	PSRAM map, repeated and mirrored at 0x00800000 offsets
0x10000000	0x1fffffff	0x0fffffff	Memory-mapped SPI chip
??????????	??????????	??????????	????????????????????????????????
0x70000000	0x7000cfff	0xcfff	On-chip SRAM (maybe cache?)
??????????	??????????	??????????	????????????????????????????????
0x80000000	0x80000008	0x08	Config block (chip version, etc.)
0x82200000	??????????	??????????	
0x83000000	??????????	??????????	
0xa0000000	0xa0000008	0x08	Config block (mirror?)
0xa0010000	??????????	??????????	(?SPI mode?) ??????????????????????
0xa0020000	0xa0020e10	0x0e10	GPIO control block
0xa0030000	0xa0030040	0x40	WDT block + 0x08 -> WDT register (?) + 0x18 -> Boot src (?)
0xa0030800	??????????	??????????	??????????????????????????????
0xa0040000	??????????	??????????	????????????????????????????????
0xa0050000	??????????	??????????	????????????????????????????????
0xa0060000	??????????	??????????	?? Possible IRQs at 0xa0060200 ????
0xa0070000	==========	==========	== Empty (all zeroes) ==========
0xa0080000	0xa008005c	0x5c	UART1 block
0xa0090000	0xa009005c	0x5c	UART2 block
0xa00a0000	??????????	??????????	????????????????????????????????

Booting an OS

From here we were able to progress rapidly on many fronts, but our goal of a port of NuttX remained elusive because there was no documentation on the interrupt controller within the canon of Shanzhai datasheets. Although we were able to find the routines that installed the interrupt handlers through static analysis of the binaries, we were unable to determine the address offsets of the interrupt controller itself.

At this point, we had to open the Mediatek codebase and refer to the include file that contained the register offsets and bit definitions of the interrupt controller. We believe this is acceptable because facts are not copyrightable. Justice O'Connor wrote in Feist v. Rural (449 U.S. 340, 345, 349 (1991). See also Sony Computer Entm't v. Connectix Corp., 203 F. 3d 596, 606 (9th Cir. 2000); Sega Enterprises Ltd. v. Accolade, Inc., 977 F.2d 1510, 1522-23 (9th Cir. 1992)) that

> "Common sense tells us that 100 uncopyrightable facts do not magically change

> their status when gathered together in one place. ... The key to resolving the tension lies in understanding why facts are not copyrightable: The sine qua non of copyright is originality"

and

> "Notwithstanding a valid copyright, a subsequent compiler remains free to use the facts contained in another's publication to aid in preparing a competing work, so long as the competing work does not feature the same selection and arrangement".

And so here, we must tread carefully: we must extract facts, and express them in our own selection and arrangement. Just as the facts that "John Doe's phone number is 555-1212" and "John Doe's address is 10 Main St." is not copyrightable, we need to extract facts such as "The interrupt controller's base address in 0xA0060000", and "Bit 1 controls status reporting of the LCD" from the include files, and re-express them in our own header files.

The situation is further complicated by blocks for which we have absolutely no documentation, not even an explanation of what the registers mean or how the blocks function. For these blocks, we reduce their initialization into a list of address and data pairs, and express this in a custom scripting language called "scriptic". We invented our own language to avoid subconscious plagiarism – it is too easy to read one piece of code and, from memory, code something almost exactly the same. By transforming the code into a new language, we're forced to consider the facts presented and express them in an original arrangement.

Scriptic is basically a set of assembler macros, and the syntax is very simple. Here is an example of a scriptic script:

```
#include "scriptic.h" #include "fernvale-pll.h" sc_new "set_plls", 1,
0, 0 sc_write16 0, 0, PLL_CTRL_CON2 sc_write16 0, 0, PLL_CTRL_CON3 sc_
write16 0, 0, PLL_CTRL_CON0 sc_usleep 1 sc_write16 1, 1, PLL_CTRL_UPLL_
CON0 sc_write16 0x1840, 0, PLL_CTRL_EPLL_CON0 sc_write16 0x100, 0x100,
PLL_CTRL_EPLL_CON1 sc_write16 1, 0, PLL_CTRL_MDDS_CON0 sc_write16 1, 1,
PLL_CTRL_MPLL_CON0 sc_usleep 1 sc_write16 1, 0, PLL_CTRL_EDDS_CON0 sc_
write16 1, 1, PLL_CTRL_EPLL_CON0 sc_usleep 1 sc_write16 0x4000, 0x4000,
PLL_CTRL_CLK_CONDB sc_usleep 1 sc_write32 0x8048, 0, PLL_CTRL_CLK_CONDC
/* Run the SPI clock at 104 MHz */ sc_write32 0xd002, 0, PLL_CTRL_CLK_
CONDH sc_write32 0xb6a0, 0, PLL_CTRL_CLK_CONDC sc_end
```

This script initializes the PLL on the MT6260. To contrast, here's the first few lines of the code snippet from which this was derived:

```
// enable HW mode TOPSM control and clock CG of PLL control *PLL_PLL_
CON2 = 0x0000; // 0xA0170048, bit 12, 10 and 8 set to 0 to enable TOPSM
control // bit 4, 2 and 0 set to 0 to enable clock CG of PLL control
*PLL_PLL_CON3 = 0x0000; // 0xA017004C, bit 12 set to 0 to enable TOPSM
control // enable delay control *PLL_PLLTD_CON0= 0x0000; //0x A0170700,
bit 0 set to 0 to enable delay control //wait for 3us for TOPSM and de-
lay (HW) control signal stable for(i = 0 ; i < loop_1us*3 ; i++); //
enable and reset UPLL reg_val = *PLL_UPLL_CON0; reg_val |= 0x0001; *PLL_
UPLL_CON0 = reg_val; // 0xA0170140, bit 0 set to 1 to enable UPLL and
generate reset of UPLL
```

The original code actually goes on for pages and pages, and even this snippet is surrounded by conditional statements which we culled as they were not relevant facts to initializing the PLL correctly.

With this tool added to our armory, we were finally able to code sufficient functionality to boot NuttX on our own Fernvale hardware.

Toolchain

Requiring users to own a Novena ROMulator to hack on Fernvale isn't a scalable solution, and thus in order to round out the story, we had to create a complete developer toolchain. Fortunately, the compiler is fairly cut-and-dry – there are many compilers that support ARM as a target, including clang and gcc. However, flashing tools for the MT6260 are much more tricky, as all the existing ones that we know of are proprietary Windows programs, and Osmocom's loader doesn't support the protocol version required by the MT6260. Thus, we had to reverse engineer the Mediatek flashing protocol and write our own open-source tool.

Fortunately, a blank, unfused MT6260 shows up as /dev/ttyUSB0 when you plug it into a Linux host – in other words, it shows up as an emulated serial device over USB. This at least takes care of the lower-level details of sending and receiving bytes to the device, leaving us with the task of reverse engineering the protocol layer. xobs located the internal boot ROM of the MT6260 and performed static code analysis, which provided a lot of insight into the protocol. He also did some static analysis on Mediatek's Flashing tool and captured live traces using a USB protocol analyzer to clarify the remaining details. Below is a summary of the commands he extracted, as used in our open version of the USB flashing tool.

```
enum mtk_commands { mtk_cmd_old_write16 = 0xa1, mtk_cmd_old_read16
= 0xa2, mtk_checksum16 = 0xa4, mtk_remap_before_jump_to_da = 0xa7,
mtk_jump_to_da = 0xa8, mtk_send_da = 0xad, mtk_jump_to_maui = 0xb7,
mtk_get_version = 0xb8, mtk_close_usb_and_reset = 0xb9, mtk_cmd_new_
read16 = 0xd0, mtk_cmd_new_read32 = 0xd1, mtk_cmd_new_write16 = 0xd2,
mtk_cmd_new_write32 = 0xd4, // mtk_jump_to_da = 0xd5, mtk_jump_to_bl =
0xd6, mtk_get_sec_conf = 0xd8, mtk_send_cert = 0xe0, mtk_get_me = 0xe1,
/* Responds with 22 bytes */ mtk_send_auth = 0xe2, mtk_sla_flow = 0xe3,
mtk_send_root_cert = 0xe5, mtk_do_security = 0xfe, mtk_firmware_version =
0xff, };
```

Current Status and Summary

After about a year of on-and-off effort between work on the Novena and Chibitronics campaigns, we were able to boot a port of NuttX on the MT6260. A minimal set of hardware peripherals are currently supported; it's enough for us to roughly reproduce the functionality of an AVR used in an Arduino-like context, but not much more. We've presented our results this year at 31C3 (slides).

The story takes an unexpected twist right around the time we were writing our CFP proposal for 31C3. The week before submission, we became aware that Mediatek released the LinkIT ONE, based on the MT2502A, in conjunction with Seeed Studios. The LinkIT ONE is directly aimed at providing an Internet of Things platform to entrepreneurs and individuals. It's integrated into the Arduino framework, featuring an open API that enables the full functionality of the chip, including GSM functions. However, the core OS that boots on the MT2502A in the LinkIT ONE is still the proprietary Nucleus OS and one cannot gain direct access to the hardware; they must go through the API calls provided by the Arduino shim.

Realistically, it's going to be a while before we can port a reasonable fraction of the MT6260's features into the open source domain, and it's quite possible we will never be able to do a blob-free implementation of the GSM call functions, as those are controlled by a DSP unit that's even more obscure and undocumented. Thus, given the robust functionality of the LinkIT ONE compared to Fernvale, we've decided to leave it as an open question to the open source community as to whether or not there is value in continuing the effort to reverse engineer the MT6260: How important is it, in practice, to have a blob-free firmware?

Regardless of the answer, we released Fernvale because we think it's imperative to exercise our fair use rights to reverse engineer and create interoperable, open

source solutions. Rights tend to atrophy and get squeezed out by competing interests if they are not vigorously exercised; for decades engineers have sat on the sidelines and seen ever more expansive patent and copyright laws shrink their latitude to learn freely and to innovate. I am saddened that the formative tinkering I did as a child is no longer a legal option for the next generation of engineers. The rise of the Shanzhai and their amazing capabilities is a wake-up call. I see it as evidence that a permissive IP environment spurs innovation, especially at the grass-roots level. If more engineers become aware of their fair use rights, and exercise them vigorously and deliberately, perhaps this can catalyze a larger and much-needed reform of the patent and copyright system.

Origins and Mission of the Federal Reserve

Ben Bernanke, The Federal Reserve and the Financial Crisis, March 2012

What I want to talk about in these four lectures is the Federal Reserve and the financial crisis. My thinking about this is conditioned by my experience as an economic historian. When one talks about the issues that occurred over the past few years, I think it makes the most sense to consider them in the broader context of central banking as it has been practiced over the centuries. So, even though I am going to focus in these lectures quite a bit on the financial crisis and how the Fed responded, I need to go back and look at the broader context. As I talk about the Fed, I will talk about the origin and mission of central banks in general; in looking at previous financial crises, most notably the Great Depression, you will see how that mission informed the Fed's actions and decisions.

In this first lecture, I will not touch on the current crisis at all. Instead, I will talk about what central banks are, what they do, and how central banking got started in the United States. I will talk about how the Fed engaged with its first great challenge, the Great Depression of the 1930s. In the second lecture, I will pick up the history from there. I will review developments in central banking and with the Federal Reserve after World War II, talking about the conquest of inflation, the Great Moderation, and other developments that occurred after 1945. But in that lecture I will also spend a good bit of time talking about the buildup to the crisis and some of the factors that led to the crisis of 2008–2009. In lecture three, I will turn to more recent events. I will talk about the intense phase of the financial crisis, its causes, its implications, and particularly the response to the crisis by the Federal Reserve and by other policymakers. And then, in the final lecture I will look at the aftermath. I will talk about the recession that followed the crisis, the policy response of the Fed (including monetary policy), the broader response in terms of the changes in financial regulation, and a little bit of forward-looking discussion about how this experience will change how central banks operate and how the Federal Reserve will operate in the future.

***

So let's talk in general about what a central bank is. If you have some background in economics you know that a central bank is not a regular bank; it is a government agency, and it stands at the center of a country's monetary and financial system. Central banks are very important institutions; they have helped to guide the development of modern financial and monetary systems and they play a major role in

economic policy. There have been various arrangements over the years, but today virtually all countries have central banks: the Federal Reserve in the United States, the Bank of Japan, the Bank of Canada, and so on. The main exception is in cases where there is a currency union, where a number of countries collectively share a central bank. By far the most important example of that is the European Central Bank, which is the central bank for seventeen European countries that share the euro as their common currency. But even in that case, each of the participating countries does have its own central bank, which is part of the overall system of the euro. Central banks are now ubiquitous; even the smallest countries typically have central banks.

What do central banks do? What is their mission? It is convenient to talk about two broad aspects of what central banks do. The first is to try to achieve macroeconomic stability. By that I mean achieving stable growth in the economy, avoiding big swings—recessions and the like—and keeping inflation low and stable. That is the economic function of a central bank. The other function of central banks, which is going to get a lot of attention in these lectures, is to maintain financial stability. Central banks try to keep the financial system working normally and, in particular, they try to either prevent or mitigate financial panics or financial crises.

What are the tools that central banks use to achieve these two broad objectives? In very simple terms, there are basically two sets of tools. On the economic stability side, the main tool is monetary policy. In normal times, for example, the Fed can raise or lower short-term interest rates. It does that by buying and selling securities in the open market. Usually, if the economy is growing too slowly or inflation is falling too low, the Fed can stimulate the economy by lowering interest rates. Lower interest rates feed through to a broad range of other interest rates, which encourages spending on the acquisition of homes, for example, and on construction, investment by firms, and so on. Lower interest rates generate more demand, more spending, and more investment in the economy, and that creates more thrust in growth. And similarly, if the economy is growing too hot, if inflation is becoming a problem, then the normal tool of central bank is to raise interest rates. Raising the overnight interest rate that the Fed charges banks to lend money, known in the United States as the federal funds rate, feeds higher interest rates through the system. This helps to slow the economy by raising the cost of borrowing, of buying a house or a car, or of investing in capi-tal goods, reducing pressure on an overheating economy. Monetary policy is the basic tool that central banks have used for many years to try to keep the economy on a more or less even keel in terms of both growth and inflation.

The main tool of central banks for dealing with financial panics or financial crises is a little less familiar: the provision of liquidity. In order to address financial stability

concerns, one thing that central banks can do is make short-term loans to financial institutions. As I will explain, providing short-term credit to financial institutions during a period of panic or crisis can help calm the market, can help stabilize those institutions, and can help mitigate or end a financial crisis. This activity is known as the "lender of last resort" tool. If financial markets are disrupted and financial institutions do not have alternative sources of funding, then the central bank stands ready to serve as the lender of last resort, providing liquidity and thereby helping to stabilize the financial system.

There is a third tool that most central banks (including the Fed) have, which is financial regulation and supervision. Central banks usually play a role in supervising the banking system, assessing the extent of risk in their portfolios, making sure their practices are sound and, in that way, trying to keep the financial system healthy. To the extent that a financial system can be kept healthy and its risk-taking within reasonable bounds, then the chance of a financial crisis occurring in the first place is reduced. This activity is not unique to central banks, however. In the United States, for example, there are a number of different agencies, such as the Federal Deposit Insurance Corporation (FDIC) and the Office of the Comptroller of the Currency, that work with the Fed in supervising the financial system. Because this is not unique to central banks, I will downplay this for the moment and focus on our two principal tools: monetary policy and lender of last resort activities.

Where do central banks come from? One thing people do not appreciate is that central banking is not a new development. It has been around for a very long time. The Swedes set up a central bank in 1668, three and a half centuries ago. The Bank of England was founded in 1694,1 and was for many decades, if not centuries, the most important and influential central bank in the world. France established a central bank in 1800. So central bank theory and practice is not a new thing. We have been thinking about these issues collectively as an economics profession and in other contexts for many years.

I need to talk a little bit about what a financial panic is. In general, a financial panic is sparked by a loss of confidence in an institution. The best way to explain this is to give a familiar example. If you have seen the movie It's a Wonderful Life, you know that one of the problems Jimmy Stewart's character runs into as a banker is a threatened run on his institution. What is a run? Imagine a situation like Jimmy Stewart's, before there was deposit insurance and the FDIC. And imagine you have a bank on the corner, just a regular commercial bank; let's call it the First Bank of Washington, D.C. This bank makes loans to businesses and the like, and it finances itself by taking deposits from the public. These deposits are called demand deposits, which means that depositors can pull out their money anytime they want, which is

important because people use deposits for ordinary activities, like shopping.

Now imagine what would happen if, for some reason, a rumor goes around that this bank has made some bad loans and is losing money. As a depositor, you say to yourself, "Well, I don't know if this rumor is true or not. But what I do know is that if I wait and everybody else pulls out their money and I'm the last person in line, I may end up with nothing." So, what are you going to do? You are going to go to the bank and say, "I'm not sure if this rumor is true or not, but, knowing that everybody else is going to pull their deposits out of the bank, I'm going to pull my money out now." And so, depositors line up to pull out their cash.

Now, no bank holds cash equal to all its deposits; it puts that cash into loans. So the only way the bank can pay off the depositors, once it goes through its minimal cash reserves, is to sell or otherwise dispose of its loans. But it is very hard to sell a commercial loan; it takes time, and you usually have to sell it at a discount. Before a bank even gets around to doing that, depositors are at the door asking, "Where is my money?" So a panic can be a self-fulfilling prophecy, leading the bank to fail; it will have to sell off its assets at a discount price and, ultimately, many depositors might lose money, as happened in the Great Depression.

Panics can be a serious problem. If one bank is having problems, people at the bank next door may begin to worry about problems at their bank. And so, a bank run can lead to widespread bank runs or a banking panic more broadly. Sometimes, pre-FDIC, banks would respond to a panic or a run by refusing to pay out deposits; they would just say, "No more; we're closing the window." So the restriction on the access of depositors to their money was another bad outcome and caused problems for people who had to make a payroll or buy groceries. Many banks would fail and, beyond that, banking panics often spread into other markets; they were often associated with stock market crashes, for example. And all those things together, as you might expect, were bad for the economy.

A financial panic can occur anytime you have an institution that has longer-term illiquid assets—illiquid in the sense that it takes time and effort to sell those loans—and is financed on the other side of the balance sheet by short-term liabilities, such as deposits. Anytime you have that situation, you have the possibility that the people who put their money in the bank may say, "Wait a minute, I don't want to leave my money here; I'm pulling it out," and you have a serious problem for the institution.

So how could the Fed have helped Jimmy Stewart? Remember that central banks act as the lender of last resort. Imagine that Jimmy Stewart is paying out the money to

his depositors. He has plenty of good loans, but he cannot change those into cash, and he has people at the door demanding their money immediately. If the Federal Reserve was on the job, Jimmy Stewart could call the local Fed office and say, "Look, I have a whole bunch of good loans that I can offer as collateral; give me a cash loan against this collateral." Then Jimmy Stewart can take the cash from the central bank, pay off his depositors, and then, so long as he really is solvent (that is, as long as his loans really are good), the run will be quelled and the panic will come to an end. So by providing short-term loans and taking collateral (the illiquid assets of the institution), central banks can put money into the system, pay off depositors and short-term lenders, calm the situation, and end the panic.

This was something the Bank of England figured out very early. In fact, a key person in the intellectual development of banking was a journalist named Walter Bagehot, who thought a lot about central banking policy. He had a dictum that during a panic central banks should lend freely to whoever comes to their door; as long as they have collateral, give them money. Central banks need to have collateral to make sure that they get their money back, and that collateral has to be good or it has to be discounted. Also, central banks need to charge a penalty interest rate so that people do not take advantage of the situation; they signal that they really need the money by being willing to pay a slightly higher interest rate. If a central bank follows Bagehot's rule, it can stop financial panics. As a bank or other institution finds that it is losing its funding from depositors or other short-term lenders, it borrows from the central bank. The central bank provides cash loans against collateral. The company then pays off its depositors and things calm down. Without that source of funds, without that lender of last resort activity, many institutions would have to close their doors and could go bankrupt. If they had to sell their assets at fire-sale discount prices, that would create further problems because other banks would also find the value of their assets going down. And so, panic—through fear, rumor, or declining asset values—could spread throughout the banking system. So it is very important to get in there aggressively. As a central banker, provide that short-term liquidity and avert the collapse of the system or at least serious stress on it.

Let's talk a little bit specifically about the United States and the Federal Reserve. The Federal Reserve was founded 1914, and concerns about both macroeconomic stability and financial stability motivated the decision of Congress and President Woodrow Wilson to create it. After the Civil War and into the early 1900s, there was no central bank, so any kind of financial stability functions that could not be performed by the Treasury had to be done privately. There were some interesting examples of private attempts to create lender of last resort functions; for example, the New York Clearing House. The New York Clearing House was a private institution; it was basically a club of ordinary commercial banks in New York City. It was called

the Clearing House because, initially, that is what it was; it served as a place where banks could come at the end of each day to clear checks against one another. But over time, clearing houses began to function a little bit like central banks. For example, if one bank came under a lot of pressure, the other banks might come together in the clearing house and lend money to that bank so it could pay its depositors. And so in that respect, they served as a lender of last resort. Sometimes, the clearing houses would all agree that they were going to shut down the banking system for a week in order to look at the bank that was in trouble, evaluate its balance sheet, and determine whether it was in fact a sound bank. If it was, it would reopen and, normally, that would calm things down. So there was some private activity to stabilize the banking system.

In the end, though, these kinds of private arrangements were just not sufficient. They did not have the resources or credibility of an independent central bank. After all, people could always wonder whether the banks were acting in something other than the public interest since they were all private institutions. So it was necessary for the United States to get a lender of last resort that could stop runs on illiquid but still solvent commercial banks.

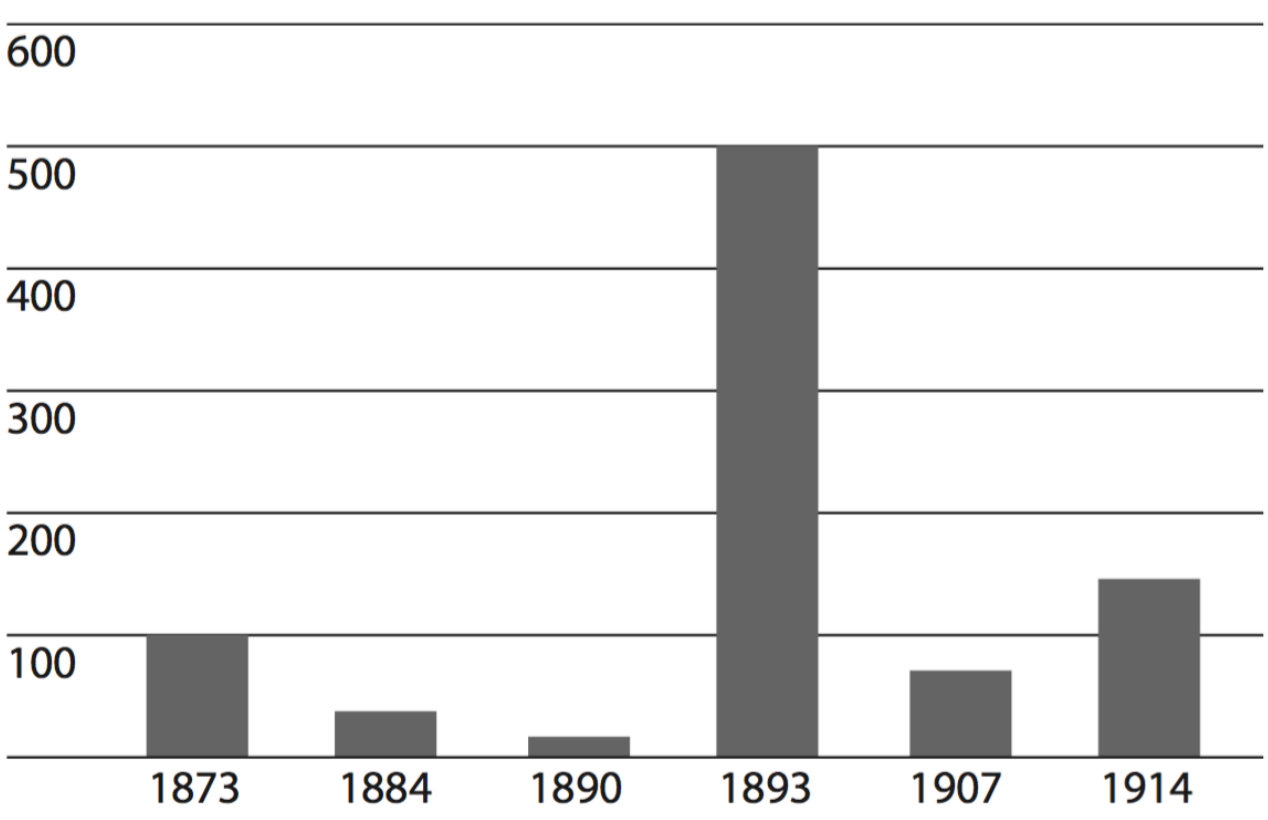


Figure 1. Bank Closings during Banking Panics, 1873–1914
Sources: For 1873–1907, Elmus Wicker, Banking Panics of the Gilded Age (New York: Cambridge University Press, 2006), table 1.3; for 1914, Federal Reserve Board, Banking and Monetary Statistics, 1914–1941.

This was not a hypothetical issue. Financial panics in the United States were a very big problem in the period from the restoration of the gold standard after the Civil War in 1879 through the founding of the Federal Reserve. Figure 1 shows the num-

ber of banks closing during each of the six major banking panics during that period in the United States.

You can see that in the very severe financial panic of 1893, more than five hundred banks failed across the country, with significant consequences for the financial system and for the economy. Fewer banks failed in the panic of 1907, but the banks that did fail were larger. After the crisis of 1907, Congress began to think that maybe they needed a government agency that could address the problem of financial panics. A twenty-three-volume study was prepared for the Congress about central banking practices, and Congress moved deliberatively toward creating a central bank. The new central bank was finally established in 1914, after yet another serious financial panic. So financial stability concerns were a major reason that Congress decided to create a central bank in the early twentieth century.

But remember that the other major mission of central banks is monetary and economic stability. For most of the period from after the Civil War until the 1930s, the United States was on a gold standard. What is a gold standard? It is a monetary system in which the value of the currency is fixed in terms of gold; for example, by law in the early twentieth century, the price of gold was set at $20.67 an ounce. So there was a fixed relationship between the dollar and a certain weight of gold, and that in turn helped set the money supply; it helped set the price level in the economy. There were central banks that helped manage the gold standard, but to a significant extent a true gold standard creates an automatic monetary system and is at least a partial alternative to a central bank.

Unfortunately, a gold standard is far from a perfect monetary system. For instance, it is a big waste of resources: you have to dig up tons of gold and move it to the basement of the Federal Reserve Bank in New York. Milton Friedman used to emphasize that a very serious cost of a gold standard was that all this gold was being dug up and then put back into another hole. But there are other more serious financial and economic concerns that practical experience has shown to be part of a gold standard.

Take the effect of a gold standard on the money supply. Since the gold standard determines the money supply, there is not much scope for the central bank to use monetary policy to stabilize the economy. In particular, under a gold standard, typically the money supply goes up and interest rates go down in periods of strong economic activity, which is the reverse of what a central bank would normally do today. Because you have a gold standard that ties the money supply to gold, there is no flexibility for the central bank to lower interest rates in a recession or raise interest rates to counter inflation. Some people view that as a benefit of the gold standard—

taking away the central bank's discretion—and there is an argument to be made for that, but it does have the side effect that there was more year-to-year volatility in the economy under the gold standard than there has been in modern times. Volatility in output variability and year-to-year movements in inflation were much greater under the gold standard.

There are other concerns with the gold standard. One of the things a gold standard does is to create a system of fixed exchange rates between the currencies of countries that are on the gold standard. For example, in 1900, the value of a dollar was about twenty dollars per ounce of gold. At the same time, the British set their gold standard at roughly four British pounds per ounce of gold. Twenty dollars equals one ounce of gold, and one ounce of gold equals four British pounds, so twenty dollars equals four pounds. Basically, one pound is valued at five dollars. So essentially, if both countries are on the gold standard, the ratio of prices between the two exchange rates is fixed. There is no variability, unlike today, when the euro can go up and down against the dollar. Again, some people would argue that is beneficial, but there is at least one problem: if there are shocks or changes in the money supply in one country and perhaps even a bad set of policies, other countries that are tied to the currency of that country will experience some of the effects.

I will give you a modern example. Today, China ties its currency to the dollar. It has become more flexible lately, but for a long time there has been a close relationship between the Chinese currency and the U.S. dollar. That means that if the Fed lowers interest rates and stimulates the U.S. economy because, say, it is in a recession, essentially monetary policy becomes easier in China as well because interest rates have to be the same in different countries with essentially the same currency. And those low interest rates may not be appropriate for China, and as a result China may experience inflation because it is essentially tied to U.S. monetary policy. So fixed exchange rates between countries tend to transmit both good and bad policies between those countries and take away the independence that individual countries have to manage their own monetary policy.

Yet another issue with the gold standard has to do with speculative attack. Normally, a central bank with a gold standard keeps only a fraction of the gold necessary to back the entire money supply. Indeed, the Bank of England was famous for keeping "a thin film of gold," as John Maynard Keynes called it. The British central bank kept only a small amount of gold and relied on its credibility in standing by the gold standard under all circumstances, so that nobody ever challenged it about that issue. But if, for whatever reason, markets lose confidence in a central bank's commitment to maintain the gold standard, the currency can become subject to a speculative attack. This is what happened to the British. In 1931, for a lot of good reasons,

speculators lost confidence that the British pound would maintain its gold convertibility, so (just like a run on the bank) they all brought their pounds to the Bank of England and said, "Give me gold." It did not take very long for the Bank of England to run out of gold because it did not have all the gold it needed to support the money supply, which essentially forced Great Britain to leave the gold standard.

There is a story that while a British Treasury official was taking a bath, an aide came running in saying, "We're off the gold standard! We're off the gold standard!" And the official said, "I didn't know we could do that!" But they could, and they had to. They had no choice because there was a speculative attack on the pound. Moreover, as we saw in the case of the United States, the gold standard was associated with many financial panics. The gold standard did not always assure financial stability.

Finally, one of the strengths that people cite for the gold standard is that it creates a stable value for the currency, it creates a stable inflation. That is true over very long periods. But over shorter periods, maybe up to five or ten years, you can actually have a lot of inflation (rising prices) or deflation (falling prices) with a gold standard because the amount of money in the economy varies according to things like gold strikes. So, for example, if gold is discovered in California and the amount of gold in the economy goes up, that will cause inflation, whereas if the economy is growing faster and there is a shortage of gold, that will cause deflation. So over shorter periods, a country on the gold standard frequently had both inflations and deflations. Over long periods—decades—prices were quite stable.

This was a very significant concern in the United States. In the latter part of the nineteenth century, there was a shortage of gold relative to economic growth, and since there was not enough gold— the money supply was shrinking relative to the economy—the U.S. economy was experiencing deflation, that is, prices were gradually falling over this period. This caused problems, particularly for farmers and people in other agriculture-related occupations. Think about this for a moment. If you are a farmer in Kansas and you have a mortgage that requires a fixed payment of twenty dollars each month, the amount of money you have to pay is fixed. But how do you get the money to pay it? By growing crops and selling them in the market. Now, if you have deflation, that means that the price of your corn or cotton or grain is falling over time, but your payment to the bank stays the same. Deflation created a grinding pressure on farmers as they saw the prices of their products going down while their debt payments remained unchanged. Farmers were squeezed by this decline in their crop prices, and they recognized that this deflation was not an accident. The deflation was being caused by the gold standard.

So William Jennings Bryan ran for president on a platform the principal plank of

which was the need to modify the gold standard. In particular, he wanted to add silver to the metallic system so that there would be more money in circulation and more inflation. But he spoke about this in the very eloquent way of nineteenth-century orators. He said, "You shall not press down upon the brow of labor this crown of thorns; you shall not crucify mankind upon a cross of gold." What he was saying is that the gold standard was killing honest, hardworking farmers who were trying to make their payments to the bank and found the price of their crops going down over time.

So the gold standard created problems and was a motivation for the founding of the Federal Reserve. In 1913, finally after all the study, Congress passed the Federal Reserve Act, which established the Federal Reserve. President Wilson viewed this as the most important domestic accomplishment of his presidency. Why did they want a central bank? The Federal Reserve Act called on the newly established Fed to do two things: first, to serve as a lender of last resort and to try to mitigate the panics that banks were experiencing every few years; and second, to manage the gold standard, that is, to take the sharp edges off the gold standard to avoid sharp swings in interest rates and other macroeconomic variables.

Interestingly, the Fed was not the first attempt by Congress to create a central bank. There had been two previous attempts, one of them suggested by Alexander Hamilton and the second somewhat later in the nineteenth century. In both cases, Congress let the central bank die. The problem was disagreement between what today we would call Main Street and Wall Street. The folks on Main Street—farmers, for example—feared that the central bank would be mainly an instrument of the moneyed interests in New York and Philadelphia and would not represent the entire country, would not be a national central bank. Both the first and the second attempts at creating a central bank failed for that reason.

Woodrow Wilson tried a different approach: he created not just a single central bank in Washington but twelve Federal Reserve banks located in major cities across the country. Figure 2 shows the twelve Federal Reserve districts (which we still have today), and each one has a Federal Reserve Bank.2 Then a Board of Governors in Washington, D.C., oversees the whole system. The value of this structure was that it created a central bank where everybody, in all parts of the country, would have a voice and where information about all aspects of our national economy would be heard in Washington—and that is, in fact, still the case. When the Fed makes monetary policy, it takes into account the views of the Federal Reserve banks around the country and thus has a national approach to making policy.

The Fed was established in 1914 and for a while life was not too bad. The 1920s, the

so-called Roaring Twenties, was a period of great prosperity in the United States. Its economy was absolutely dominant in the world at that time because most of Europe was still in ruins from World War I. There were lots of new inventions. People gathered around the radio, and automobiles became much more available. There were a lot of new consumer durables and a lot of economic growth during the 1920s. So the Fed had some time to get its feet wet and establish procedures.

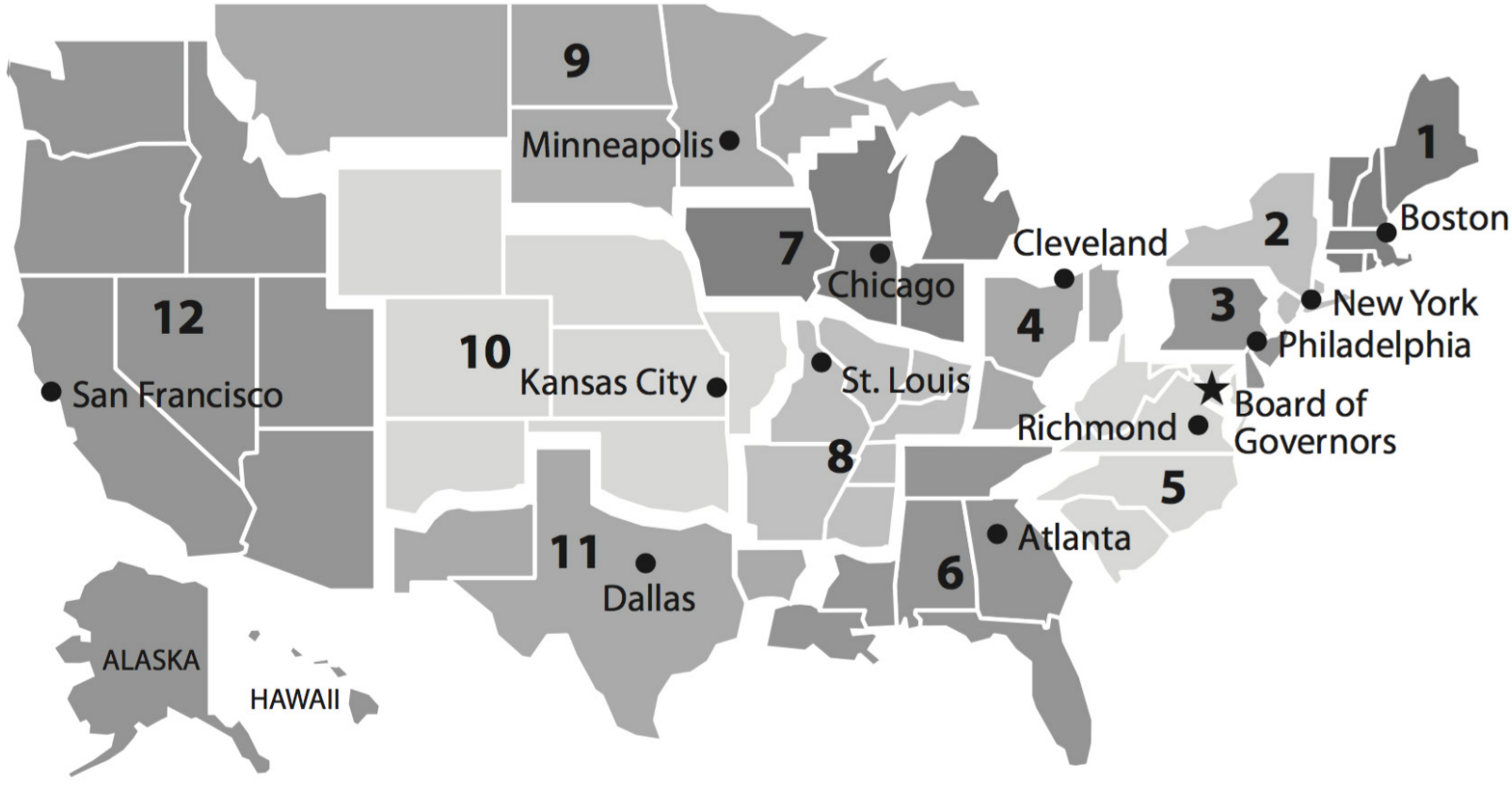


Figure 2. Federal Reserve Regions and Locations of Federal Reserve Banks and Board of Governors

Unfortunately, in 1929 the world was hit by the first great challenge to the Federal Reserve, the Great Depression. The U.S. stock market crashed on October 29th, and the financial crisis of the Great Depression was not just a U.S. phenomenon: it was global. Large financial institutions collapsed in Europe and other parts of the world. Perhaps the most damaging financial collapse was of the large Austrian bank called the Credit-Anstalt in 1931, which brought down many other banks in Europe. The economy contracted very sharply and the Depression lasted for what seems like an incredibly long time, from 1929 until 1941, when the United States entered the war following the attack on Pearl Harbor.

It is important to understand how deep and severe the Depression was. Figure 3 shows the stock market, and you can see at the left a vertical line showing October 1929, a very sharp decline in stock prices. This was the crash that was made famous by many writers including John Kenneth Galbraith and others, who told colorful stories about brokers jumping out of windows. But what I want you to take from this picture is that the crash of 1929 was only the first step in what was a much more serious decline. You see how stock prices kept falling, and by mid-1932 they

had fallen an incredible 85 percent from their peak. So this was much worse than just a couple of bad days in the stock market.

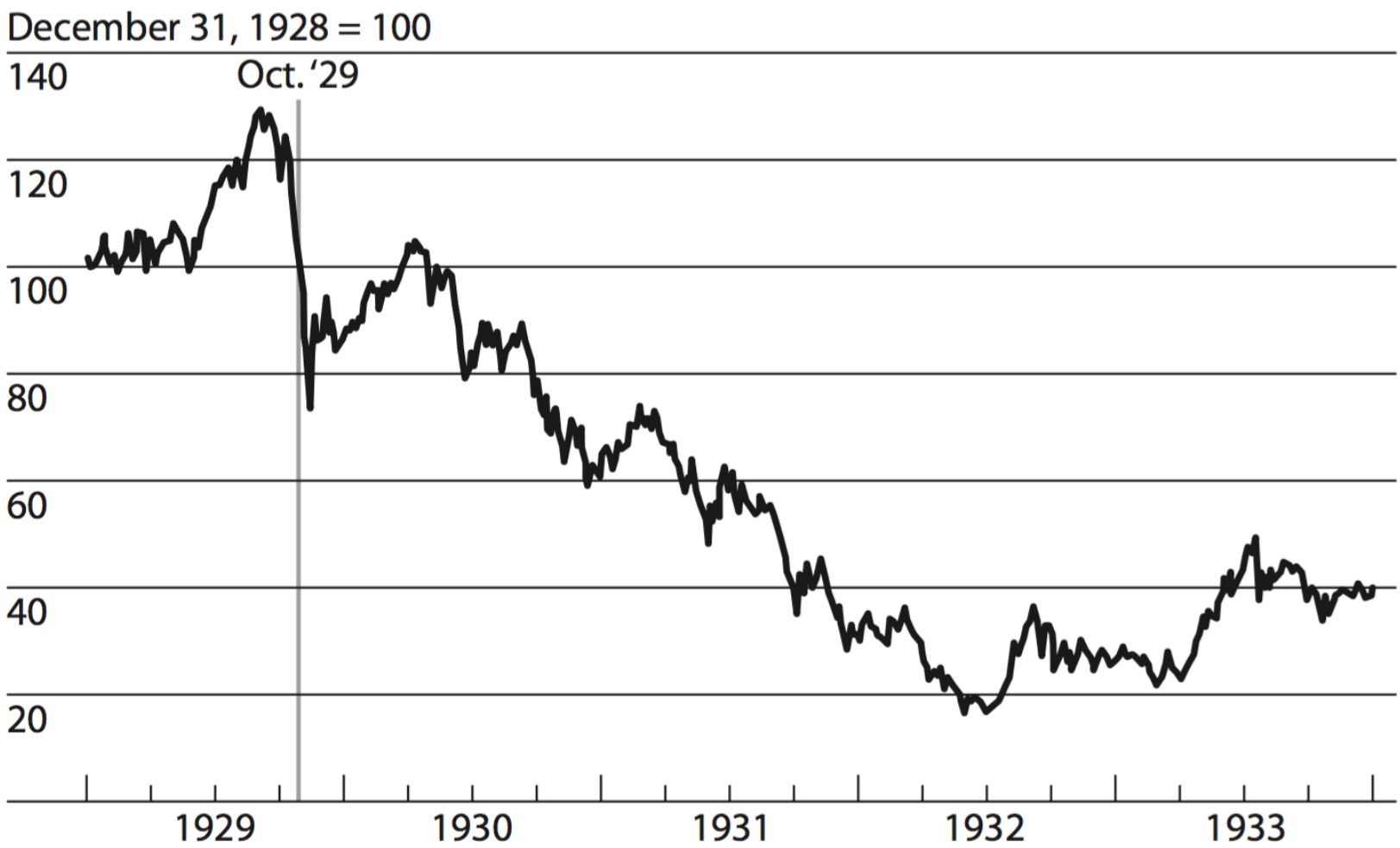


Figure 3. S&P Composite Equity Price Index, 1929–1933
Source: Center for Research in Securities Prices, Index File on the S&P 500

The real economy, the nonfinancial economy, also suffered very greatly. Figure 4a shows growth in real GDP. If the bar is above the zero line, it is a growth period. If it is below, it is a contraction period. In 1929, the economy grew by more than 5 percent and was still growing very substantially. But you can see that from 1930 to 1933, the economy contracted by very large amounts every year. So it was an enormous contraction of GDP, close to one-third overall between 1929 and 1933. At the same time, the economy was experiencing deflation (falling prices). And as you can see in figure 4b, in 1931 and 1932 prices fell by about 10 percent. So if you were a farmer who had had difficulty in the late nineteenth century, imagine what is happening to you in 1932, when crop prices are dropping by half or more and you still have to make the same payment to the bank for your mortgage.

As the economy contracted, unemployment soared. We did not have the same survey of individual households in the 1930s that we have today, and so the numbers in figure 5 are estimated; they are not precise numbers. But as best we can tell, at its peak in the early 1930s, unemployment approached 25 percent. The shaded area is the recession period. Even at the end of the 1930s, before the war changed everything, unemployment was still around 13 percent.

As you might guess, with all that was going wrong in the economy, a lot of depositors ran on their banks and many banks failed. Figure 6 shows the number of bank failures in each year, and you can see an enormous spike in the early 1930s.

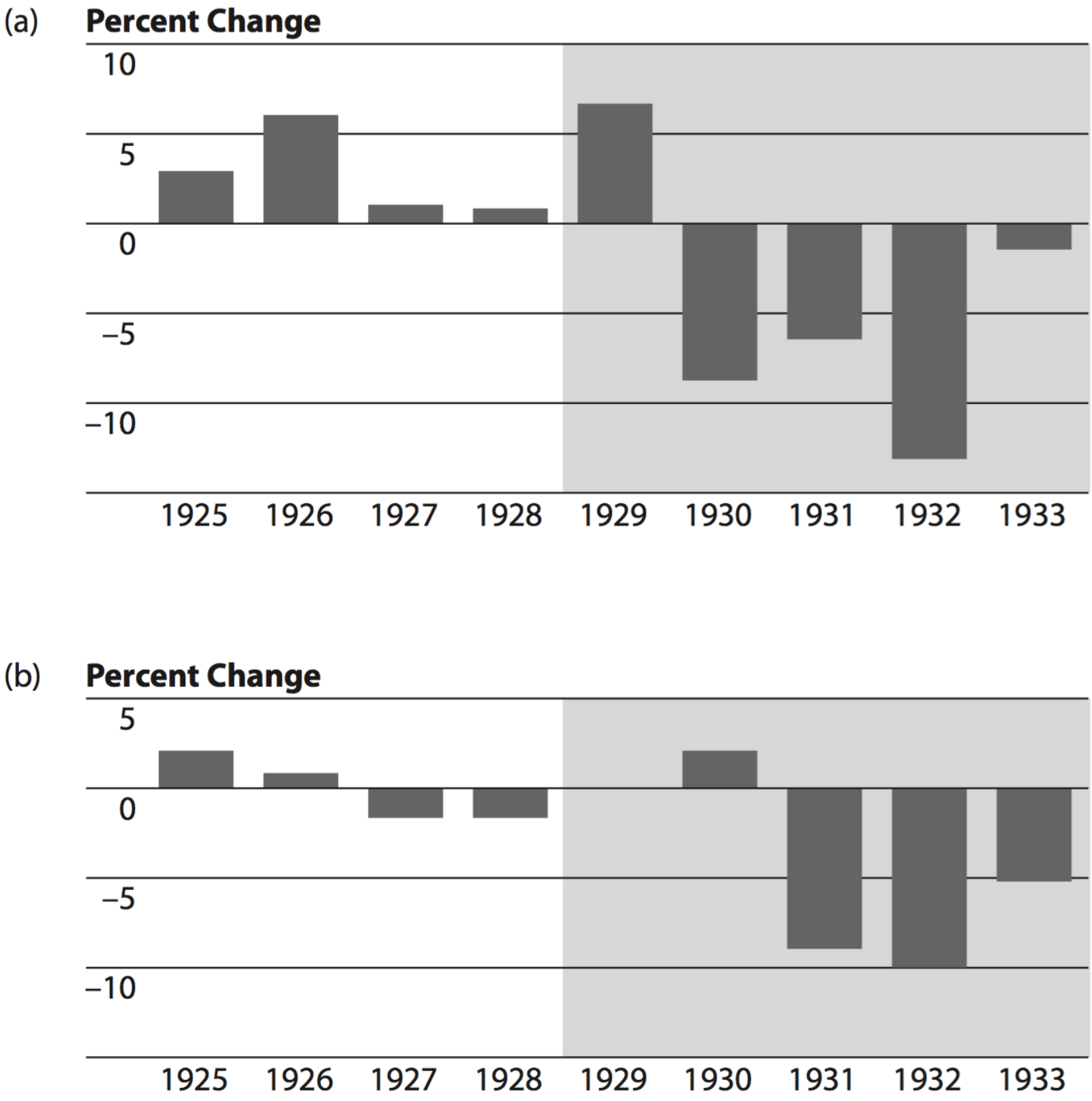


Figure 4a. Real GDP, 1925–1934
Note: Shading represents years of the Great Depression.
Source: Historical Statistics of the United States, Millennial Edition (New York: Cambridge University Press, 2006), table Ca9.

Figure 4b. Consumer Price Index, 1925–1934
Source: Historical Statistics of the United States, Millennial Edition, online, table Cc1.

What caused this colossal calamity (which, I reiterate, was not just a U.S. problem but a global problem)? Germany had a worse depression than the United States, and that led more or less directly to the election of Hitler in 1933. So what happened?

What caused the Great Depression? This is a tremendously important subject and has received a lot of attention from economic historians, as you might imagine. And as often is the case for very large events, there were many different causes. A few are the repercussions of World War I; problems with the international gold standard, which was being reconstructed but with a lot of problems after World War I; the famous bubble in stock prices in the late 1920s; and the financial panic that spread throughout the world. So a number of factors caused the Depression. Part of the problem was intellectual—a matter of theory rather than policy per se. In the 1930s, there was a lot of support for a way of thinking about the economy called the liquidationist theory, which posited that the 1920s had been too good a time: the economy had expanded too fast; there had been too much growth; too much credit had been extended; stock prices had gone too high. What you need when you have had a period of excess is a period of deflation, a period when all the excesses are squeezed out. This theory held that the Depression was unfortunate but necessary. We had to squeeze out all of the excesses that had accumulated in the economy in the 1920s. There is a famous statement by Andrew Mellon, who was Herbert Hoover's secretary of the Treasury: "Liquidate labor, liquidate stocks, liquidate the farmers, liquidate real estate." It sounds pretty heartless and I think it was, but what he was trying to convey was that we had to get rid of all of the excesses of the 1920s and bring the country back to a more fundamentally sound economy.

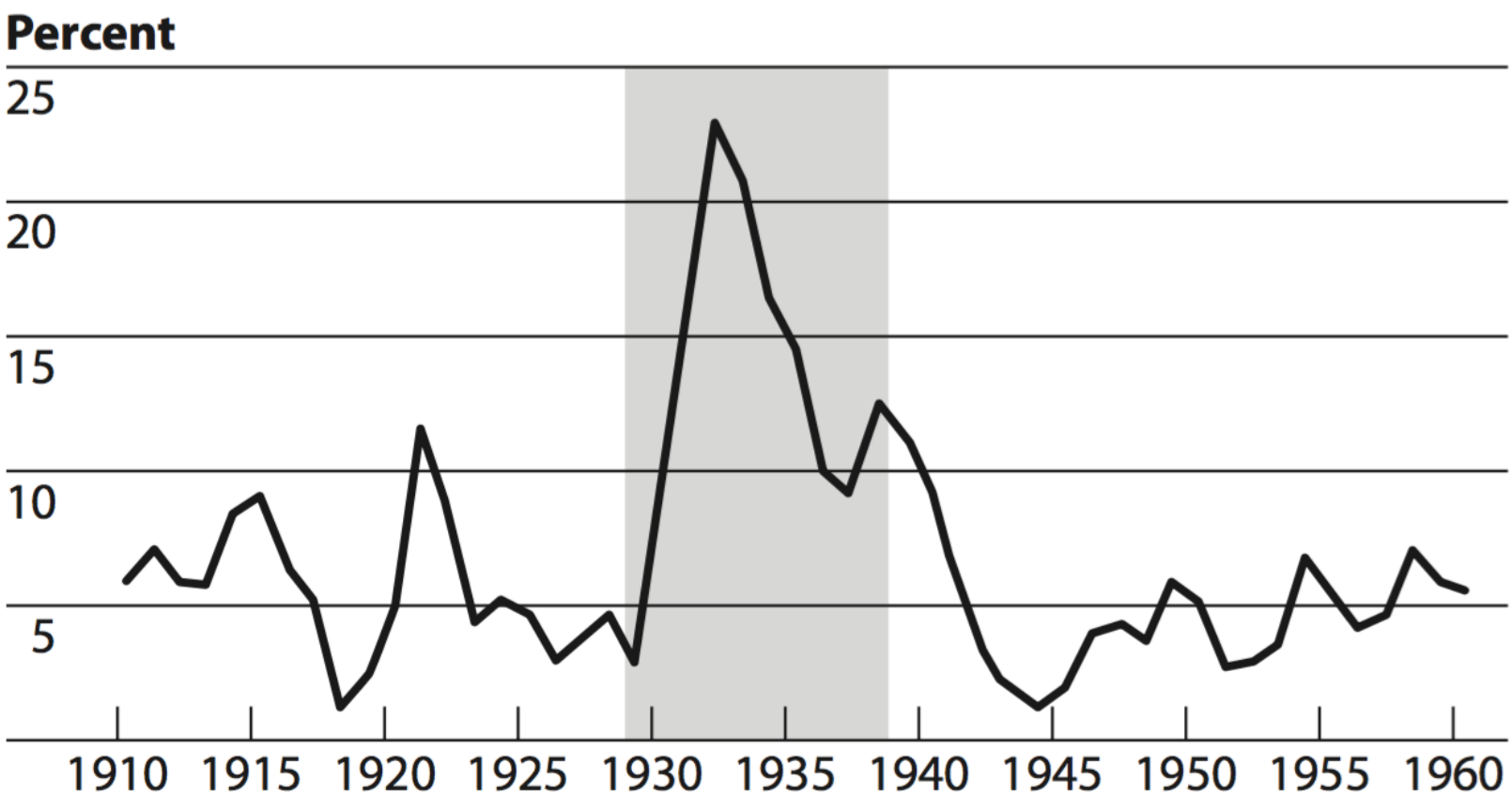


Figure 5. Unemployment Rate, 1910–1960
Note: Shading represents years of the Great Depression.
Source: Historical Statistics of the United States, Millennial Edition, table Ba475.

What was the Fed doing during this period? Unfortunately, when the Fed confront-

ed its first major challenge in the Great Depression, it failed both on the monetary policy side and on the financial stability side. On a monetary policy side, the Fed did not ease monetary policy as you would expect it to in a period of deep recession, for a variety of reasons: because it wanted to stop stock market speculation, because it wanted to maintain the gold standard, because it believed in the liquidationist theory. And so we did not get the offset to the decline that monetary policy could have provided. And indeed, what we saw was sharply falling prices. You can argue about causes of the decline in output and employment, but when you see 10 percent declines in the price level, you know monetary policy is much too tight. So deflation was an important part of the problem because it bankrupted farmers and others who relied on selling products to pay fixed debts. To make things even worse, as I mentioned earlier, if you have a gold standard, then you have fixed exchange rates. So the Fed's policies were essentially transmitted to other countries, which therefore also essentially came under excessively tight monetary policy and that contributed to the collapse. As I mentioned, one reason the Fed kept money tight was because it was worried about a speculative attack on the dollar. Remember that the British had faced that situation in 1931. The Fed was worried that there would be a similar attack that would drive the dollar off the gold standard. So, to preserve the gold standard, the Fed raised interest rates rather than lower them. They argued that keeping interest rates high would make U.S. investments attractive and prevent money from flowing out of the United States. But that was the wrong thing to do relative to what the economy needed. In 1933, Franklin Roosevelt abandoned the gold standard, and suddenly monetary policy became much less tight and there was a very powerful rebound in the economy in 1933 and 1934.

The other part of the Fed's responsibility is to be lender of last resort. And once again, the Fed did not read its mandate. It responded inadequately to the bank runs, essentially allowing a tremendous decline in the banking system as many banks failed. And as a result, bank failures swept the country. A very large fraction of the nation's banks failed; almost ten thousand banks failed in the 1930s. That continued until deposit insurance was created in 1934. Now, why did the Fed not act more aggressively as lender of last resort? Why didn't it lend to these failing banks? Well, in some cases, the banks were really insolvent. There was not much that could be done to save them. They had made loans in agricultural areas and their loans were all going bad because of the crisis in the agricultural sector. But part of it was the Fed appeared, at least to some extent, to agree with the liquidationist theory, which said that there was too much credit; the country was overbanked; let the system contract; that was the healthy thing to do. But unfortunately, that was not the right prescription.

In 1933, Franklin Roosevelt came to power. Roosevelt had a mandate to do some-

thing about the Depression. He took a variety of actions; he was very experimental. Some of those actions were quite unsuccessful. For example, something called the National Recovery Act tried to fight deflation by requiring firms to keep their prices high. But that was not going to help without a bigger money supply. So a lot of things Roosevelt tried did not work, but he did two things that I would argue did a lot to offset the problems the Fed created. The first was the establishment of deposit insurance, the FDIC, in 1934. After that, if you were an ordinary depositor and the bank failed, you still got your money back and therefore there was no incentive to run on the banks. And in fact, once deposit insurance was established, we went from literally thousands of bank failures annually to zero. It was an incredibly effective policy. The other thing FDR did was he abandoned the gold standard. And by abandoning the gold standard, he allowed monetary policy to be released and allowed expansion of the money supply, which ended the deflation and led to a powerful short-term rebound in 1933 and 1934. So the two most successful things that Roosevelt did were essentially offsetting the problems that the Fed created or at least exacerbated by not fulfilling its responsibilities.

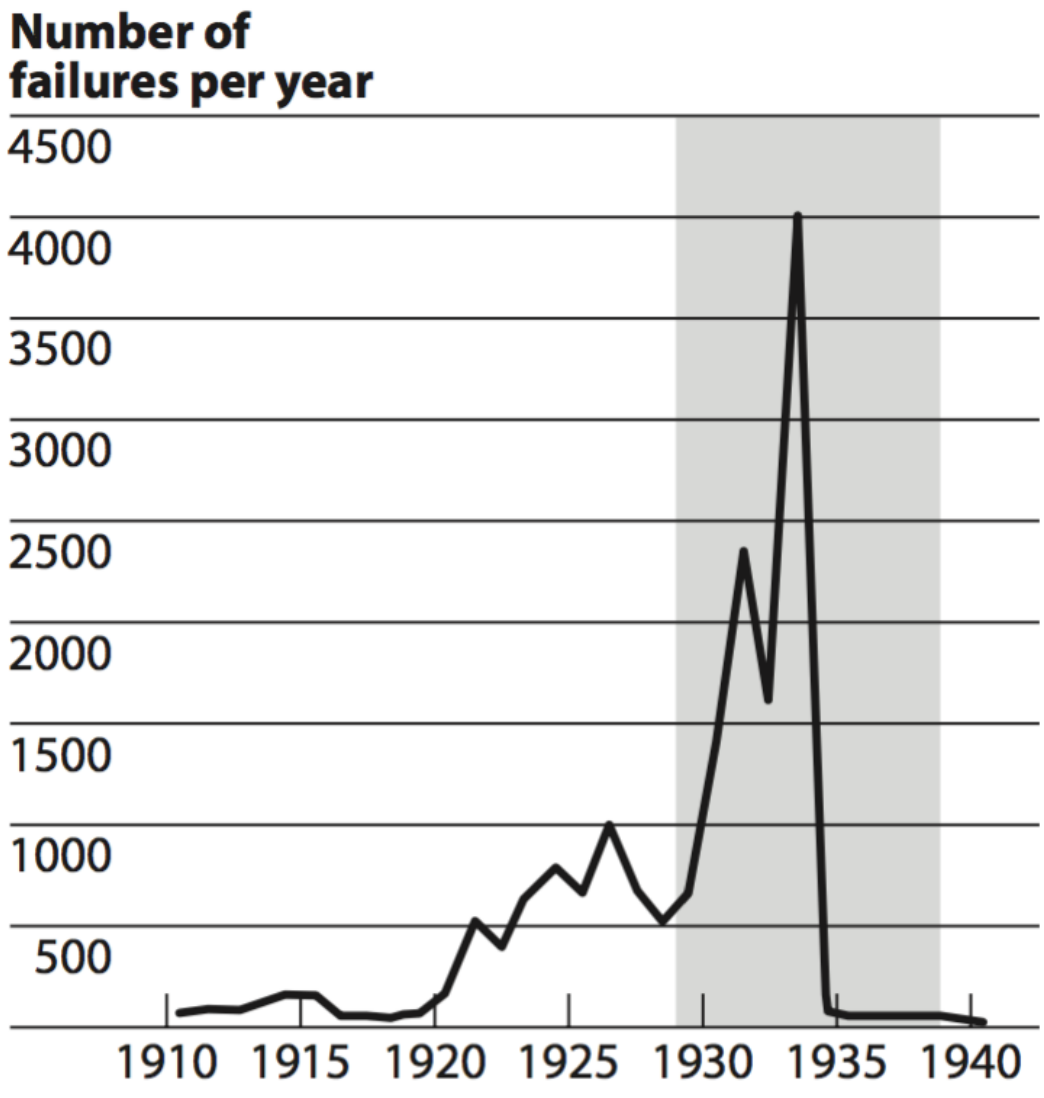


Figure 6. Bank Failures, 1910–1940
Source: Federal Reserve Board, Banking and Monetary Statistics, 1914–1941, table 66

So, what are the policy lessons? It was a global depression that had many causes, and the whole story requires you to look at the whole international system. But pol-

icy errors in the United States, as well as abroad, did play an important role. And in particular, the Federal Reserve failed in this first challenge in both parts of its mission. It did not use monetary policy aggressively to prevent deflation and the collapse in the economy, so it failed in its economic stability function. And it did not adequately perform its function as lender of last resort, allowing many bank failures and a resulting contraction in credit and also in the money supply. So the Fed did not fulfill its mission in that respect. These are key lessons, and we want to keep these in mind as we consider how the Fed responded to the 2008–2009 financial crisis.

Student: You mentioned the tightening of monetary policy in 1928 and 1929 to stem stock market speculation. Do you think that the Federal Reserve should have taken different actions, such as increasing margin requirements, to stem the speculation or was it wrong for them to take any action at all against the bubble?

Chairman Bernanke: That is a good question. The Fed was very concerned about the stock market and believed that it was excessively priced, and there was evidence for that. But they attacked it solely by raising interest rates without paying attention to the effect on the economy. So they wanted to bring down the stock market by raising interest rates, and of course they succeeded! But raising interest rates had major side effects on the economy as well. So, yes, we have learned that asset price bubbles are dangerous and we want to address them if possible, but when you can address them through financial regulatory approaches, that is usually a more pinpoint approach than just raising interest rates for everything. So margin requirements are at least looking at the variety of practices. There were a lot of very risky practices by brokers in the 1920s; it was the equivalent of day traders. Every paper boy had a hot tip for you and there were not many checks and balances on trading, who can make a trade, what margin requirements were, and so on. I think the first line of attack should have been more focused on bank lending, on financial regulation, and on the functioning of the exchanges.

Student: I have a question on the gold standard. Given everything we know about monetary policy now and about the modern economy, why is there still some argument for returning to the gold standard, and is it even possible?

Chairman Bernanke: The argument has two parts. One is the desire to maintain "the value of the dollar," that is, to have very long-term price stability. The argument is that paper money is inherently inflationary, but if we had a gold standard we would not have inflation. As I said, that is true to some extent over long periods of time. But on a year-to-year basis it is not true, and so looking at history is helpful there. The other reason advocates want to see a return to the gold standard, I think, is that

it removes discretion: it does not allow the central bank to respond with monetary policy, for example to booms and busts, and the advocates of the gold standard say it is better not to give that flexibility to a central bank.

I think, though, that the gold standard would not be feasible for both practical reasons and policy reasons. On the practical side, it is just a simple fact that there is not enough gold to meet the needs of a global gold standard, and obtaining that much gold would cost a lot. But more fundamentally, the world has changed. The reason the Bank of England could maintain the gold standard even though it had very little gold reserves was that everybody knew that the Bank's first, second, third, and fourth priorities were staying on the gold standard and that it had no interest in any other policy objective. But once there was concern that the Bank of England might not be fully committed, then there was a speculative attack that drove it off gold. Now, economic historians argue that after World War I, labor movements became much stronger and there was a lot more concern about unemployment. Before the nineteenth century people did not even measure unemployment, and after World War I you began to get much more attention to unemployment and business cycles. So in the modern world, commitment to the gold standard would mean swearing that under no circumstances, no matter how bad unemployment got, would we do anything about it using monetary policy. And if investors had 1 percent doubt that we would follow that promise, then they would have an incentive to bring their cash and take out gold, and it would be a self-fulfilling prophecy. We have seen that problem with various kinds of fixed exchange rates that have come under attack during the financial crisis. So I understand the impulse, but if you look at history you will see that the gold standard did not work very well, and it worked particularly poorly after World War I. Indeed, there is evidence that the gold standard was one of the main reasons the Depression was so deep and long. And a striking fact is that countries that left the gold standard early and gave themselves flexibility on monetary policy recovered much more quickly than the countries that stayed on gold to the bitter end.

Student: You mentioned that President Roosevelt used deposit insurance to help end bank runs and also abandoned the gold standard to help end deflation. I believe that in 1936 and 1937, up until 1941, we had a double dip and the recession went on. As you have said, today we are sort of out of the recession. What things do you think we need to be careful of—things that possibly were mistakes made in the Great Depression that we should avoid today?

Chairman Bernanke: Right, it is not generally appreciated that the Great Depression actually was two recessions. There was a very sharp recession in 1929 to 1933; from 1933 to 1937, there was actually a decent amount of growth and the stock market

recovered some; but in 1937 to 1938, there was a second recession that was not quite as serious as the first one but was still serious. There is a lot of controversy about it, but one view that was advanced early on was that the second recession came from a premature tightening of monetary and fiscal policy. In 1937 and 1938, Roosevelt was under a lot of pressure to reduce budget deficits and tighten fiscal policy. The Fed, worried about inflation, tightened monetary policy. I do not want to claim it is that simple—a lot was happening—but the early interpretation was that the reversal in policy was premature, which prevented the recovery from proceeding faster. If you accept that traditional interpretation, you need to be attentive to where the economy is and not move too quickly to reverse the policies that are helping the recovery.

Student: Based on a few of the graphs we saw today and other historical trends, it seems that after an economic slump, recovery often takes five or more years, as represented by the Great Depression and the oil crisis in the 1970s. Do you think it is common for unemployment to remain at high levels until sometimes a half decade after an economic slump, and that criticisms are often premature? And how do you address these concerns in a political environment where short-term fixes rule the day?

Chairman Bernanke: Well, the Depression was an extraordinary event. There were many serious declines in economic activity in the nineteenth century, but nothing quite as deep or as long as the Great Depression. The high unemployment that lasted from 1929 until basically World War II was unusual. So we should not conclude that was a normal state of affairs. Now, more generally, some research suggests that following a financial crisis it may take longer for the economy to recover because you need to restore the health of the financial system. Some argue that may be one reason this most recent recovery is not proceeding faster than it is. But I think it is still an open question and there is a lot of discussion about that research. So no, it is not always the case. If you look at recessions in the postwar period in the United States, you see that recoveries very frequently take only a couple of years—recessions are typically followed by a faster recovery. What may be different about this episode—and again this is a subject of debate—is that, unlike the other recessions in the postwar period, this one was related to and triggered by a global financial crisis, and that may be why it is already taking longer for the economy to recover.

Student: Since you said depressions are global recessions, shouldn't there be more global cooperation and shouldn't central banks have a uniform type of fix they cooperate on, instead of every country turning to its own fix?

Chairman Bernanke: The Fed and the central banks did cooperate, and continue to cooperate. One of the problems in the Depression was the bad feelings left over

from World War I. In the nineteenth century there was a reasonable amount of cooperation among central banks, but in the 1920s, Germany was facing having to pay reparations, and France, England, and the United States were all bickering about war debts, and so the politics was quite bad internationally and that impeded cooperation among the central banks. Also, international central bank cooperation is probably even more important when you have fixed exchange rates. In the 1920s, you had fixed exchange rates because of the gold standard; that meant that monetary policy in one country affected everybody. That was certainly a case for more coordination, but it did not happen. At least today we have flexible exchange rates, which can adjust and tend to insulate other countries from the effects of monetary policy in a given country, so that reduces the need for coordination somewhat—but there is still, I think, a need for coordination.